Vrije Universiteit Brussel
Faculty of Sciences
Department of Computer Science
Programming Technology Lab

Progressive Mobility

Ph.D. Dissertation

Luk Stoops

August 27, 2004

Supervisor: Prof. Dr. Theo D'Hondt
Co-supervisor: Prof. Dr. Tom Mens

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Toegepaste Wetenschappen

Acknowledgments

I thank Prof. Dr. Theo D'Hondt for supervising this dissertation and for his trust and support during my doctoral research at the Programming Technology Lab.

I especially thank my co-supervisor Prof. Dr. Tom Mens for his invaluable guidance and support that made this work possible.

I thank Dirk Deridder, Dr. Tom Tourwe and many other members of the Programming Technology Lab for proofreading the dissertation.

I thank all the other members of the Programming Technology Lab to welcome me in the team and for being great colleagues.

Special thanks to Karsten Verelst for sharing his Borg expertise.

Also thanks to the master and graduate students Julian Doan, Christian Devalez and Karl Evenepoel for joining me in this research domain and the valuable work and results they obtained during their thesis research.

Finally, I thank my parents to spark my passion for science, my friends for their support and my wife Gert and my sons Natan, Elias and Joran for putting things in perspective.

Luk Stoops

This work is made possible by one of the e-VRT projects funded by the Flemish government, a joint collaboration between Vlaamse Radio en Televisie (VRT, public broadcaster of Flanders), IMEC (Interuniversity MicroElectronics Center) and Vrije Universiteit Brussel (VUB).

Title

Progressive Mobility

Thesis

Progressive Mobility
Hides Network Latency

Table of Contents

Acknowledgments 2
Title 3
Thesis 3
Table of Contents 4
List of Figures 8
List of Tables 10
Samenvatting 11
Summary 17
1 Introduction 23
1.1 Thesis Motivation 24
1.2 Latency 25
1.3 Application availability 26
1.4 Research Goals 27
1.5 Research restrictions 28
1.6 Chapter Summaries 30
2 A Conceptual Framework for Progressive Mobility 33
2.1 Network 35
2.1.1 Architecture 35
2.1.2 Packet Switching 35
2.1.3 Data Rate 36
2.1.4 Delays in Computer Networks 36
2.1.4.1 Delays in Connection-oriented Networks 37
2.1.4.2 Delays in Connectionless Networks 37
2.1.5 Performance 38
2.1.6 Window of Opportunity 39
2.2 Application 39
2.2.1 Internal Structure 39
2.2.2 Size 40
2.2.3 Granularity 40
2.2.4 Evaluation Time 41
2.2.5 Delay 41
2.2.6 User Interfaces 41
2.2.7 Predictability 42
2.2.8 Components 42
2.2.9 Distributed Systems and Applications 43
2.2.10 Choosing an Experimental Programming Environment 43
2.2.11 Programming Languages Used 44
2.2.11.1 Borg 44
2.2.11.2 Java 45
2.2.11.3 Smalltalk 46
2.3 Techniques 48
2.3.1 Mobility 48
2.3.1.1 Host Mobility 48
2.3.1.2 Evaluator Mobility 49
2.3.1.3 Data Mobility 49
2.3.1.4 Code Mobility 49

2.3.1.5 Process Mobility 52
2.3.1.6 Weak and Strong Mobility 53
2.3.2 Server - Push versus Client - Pull 54
2.3.3 Parallelism 54
2.3.4 Reflection 55
2.3.5 Compression 55
2.3.6 Reordering and Pre-fetching 56
2.3.6.1 Profiling 56
2.3.6.2 Class File Splitting and Pre-fetching 57
2.3.6.3 Non-Strict Execution for Mobile Programs 58
2.3.7 Progressive techniques 59
2.3.8 Other related techniques 60
3 Progressive Anticipative Mobility using Pre-fetching of Permuted Code 63
3.1 Abstract 64
3.2 Introduction 65
3.3 Basic Observations, Assumptions and Restrictions 67
3.4 Profiling and Reordering 67
3.5 Reordering Algorithm 69
3.6 Pre-fetching 70
3.7 Experiment to Hide Network Latency 72
3.8 Results 75
3.8.1 Benchmark 75
3.8.2 CoolImage 78
3.8.3 Gremlin 80
3.8.4 Adapted Gremlin 81
3.9 Discussion 83
3.9.1 Speedup 83
3.9.2 Application Speedup versus Data rate 83
3.9.3 Pre-fetching Guidelines 84
3.9.4 Dealing with Semaphores 84
3.9.5 Applicability in other Environments 85
3.10 Summary and Conclusion 85
3.10.1 Network 87
3.10.2 Application 87
3.10.3 Techniques 88
4 Progressive Mobility using Component Streams 89
4.1 Abstract 90
4.2 Introduction 91
4.3 Proposed Technique 93
4.3.1 Basic Observations, Assumptions and Restrictions 93
4.3.2 Technique Description 94
4.3.2.1 Component Migration Time 95
4.3.2.2 Component Idle Time 97
4.3.2.3 Necessary Conditions for Removing Network Latency 98
4.3.3 Migration Strategies 99
4.3.3.1 Self Triggered after Last Instruction 99
4.3.3.2 Self Triggered based on Profiling 100
4.3.3.3 Under Control of a Supervisor 100
4.3.3.4 Fixed Migration Strategy 101
4.3.3.5 Dynamic Migration Strategy 101

4.3.4 Discussion 101
4.4 Experiment to Hide Network Latency 102
4.4.1 Borg Environment 102
4.4.1.1 Implementation 103
4.4.1.2 Results 105
4.4.1.3 Discussion 105
4.4.2 Java Environment 105
4.4.2.1 Implementation 106
4.4.2.2 Results 107
4.4.2.3 Discussion 108
4.5 Experiment to Reduce System Latency in Low Data Rate Environments 109
4.5.1 Implementation 109
4.5.1.1 Finding the Size of Objects 109
4.5.1.2 A Data rate Simulating OutputStream 110
4.5.2 Adding a Graphical User Interface 110
4.5.3 Strategies 111
4.5.4 Results 114
4.5.4.1 Time Needed to Finish the Application 114
4.5.4.2 Time Needed to Send the Components 116
4.5.4.3 Time Needed for the First Draw 116
4.5.4.4 Time Gained in Comparison to Normal Sending 118
4.5.5 Discussion 119
4.6 Experiment to Reduce System Latency by Parallel Evaluation 120
4.6.1 Process Migration with Opentalk 120
4.6.2 Using BOSS 121
4.6.3 Limitations in Current Smalltalk Environment 121
4.6.3.1 Passing by Value 121
4.6.3.2 Order of Object Instantiation 121
4.6.3.3 GraphicsHandles Cannot be Stored by BOSS 121
4.6.4 Experiment setup 122
4.6.4.1 Calculate and draw the fractal, migrate afterwards 122
4.6.4.2 Start CalcProcess and migrate the DrawProcess 122
4.6.4.3 Migrate the CalcProcess first, then the DrawProcess 122
4.6.5 Implementation 122
4.6.5.1 Strategy without parallel processing 122
4.6.5.2 Strategy with parallel processing 123
4.6.5.3 Results 123
4.6.6 Discussion 123
4.7 Design Guidelines 123
4.7.1 Necessary Conditions for Removing Network Latency 123
4.7.2 Guidelines 124
4.8 Summary and Conclusion 125
4.8.1 Network 126
4.8.2 Application 127
4.8.3 Techniques 127
5 Progressive Anticipative Mobility using Proactive Migration 129
5.1 Abstract 130
5.2 Introduction 131
5.3 Proposed Technique 132
5.3.1 Basic Observations, assumptions and restrictions 132

5.3.2 Technique Description ... 133
5.3.2.1 Computing the Delta ... 135
5.4 Experiment to Calculate the Delta ... 138
5.5 Results ... 139
5.6 Discussion ... 139
5.6.1 Gain for the Factorial Example ... 139
5.6.2 Applications without Implicit Stack Operations ... 140
5.6.3 Hardware Support ... 140
5.6.4 Dealing with Large Deltas ... 141
5.7 Summary and Conclusion ... 143
5.7.1 Network ... 144
5.7.2 Application ... 144
5.7.3 Techniques ... 145
6 Conclusion ... 147
6.1 Wrap-up ... 148
6.2 Results ... 148
6.3 Discussion ... 150
6.4 Future Work ... 151
6.4.1 Evaluate the Themes with other Criteria ... 151
6.4.2 Other Topics Related to Pre-fetching of Permuted Code ... 152
6.4.3 Pre-fetching of Permuted Code with Multi Node Hopping ... 152
6.4.4 Architectural Transformations to make Applications Streamable ... 154
6.4.5 Progressive Mobility using Proactive Migration and evaluation ... 155
6.4.5.1 Proposed Technique ... 155
6.4.5.2 Assumptions and Restrictions ... 156
6.4.5.3 Handling Semaphores ... 157
6.4.6 Aspects ... 158
6.4.7 New Research Projects ... 159
Bibliography ... 161

List of Figures

Figure 1: OSI and TCP/IP reference model ... 35
Figure 2: A stream of packets ... 36
Figure 3: Time needed to transport character arrays of different sizes in a TCP/IP network.. 38
Figure 4: Network protocol overhead ... 39
Figure 5: Typical entities of a running application ... 40
Figure 6: Entities subject to migration ... 48
Figure 7: Client-Server Paradigm ... 49
Figure 8: Code-on-Demand Paradigm ... 51
Figure 9: Remote Invocation Paradigm ... 51
Figure 10: Mobile Agent Paradigm ... 52
Figure 11: Splitted classes ... 57
Figure 12: Class pre-fetching ... 58
Figure 13: Permuting the source code ... 68
Figure 14: Smalltalk dependency graph ... 69
Figure 15: Normal weak code loading ... 70
Figure 16: Progressive anticipative mobility using pre-fetching of permuted code ... 71
Figure 17: Pre-fetching of permuted code cycle in JIT environment ... 72
Figure 18: Percentage of code visited before the appearance of the graphical user interface . 74
Figure 19: Parallel evaluation Benchmark @ 2400 bps ... 76
Figure 20: Parallel evaluation Benchmark @ 114 kbps ... 76
Figure 21: Parallel evaluation Benchmark @ 41 Mbps ... 77
Figure 22: Average timing results for Benchmark application ... 78
Figure 23: Timing results for CoolImage application ... 79
Figure 24: Parallel evaluation CoolImage @ 114 kbps ... 80
Figure 25: Timing results for Gremlin application ... 81
Figure 26: Timing results for adapted Gremlin application ... 82
Figure 27: Time needed to build GUI compared with original time ... 83
Figure 28: Time needed to end the application compared with the original time ... 84
Figure 29: Components of an application during the streaming phase. ... 95
Figure 30: Migration time for 1KiB code and $I_{tot} = 10^6$ ... 97
Figure 31: Component idle time versus evaluation clock speed ... 98
Figure 32: Component idle time for 100 sec application ... 98
Figure 33: Migrate after last instruction ... 100
Figure 34: Self triggered migration based on profiling ... 100
Figure 35: Migrate under control of a supervisor ... 101
Figure 36: Cooperating agents ... 102
Figure 37: Components during the Migration Process ... 102
Figure 38: Components during the Streaming Process ... 103
Figure 39: Borg proof of concept code ... 104
Figure 40: proof of concept setup ... 105
Figure 41: Java experiment architecture ... 107
Figure 42: Graphical user interface and thread windows ... 111
Figure 43: Java experiment with GUI architecture ... 112
Figure 44: Sequence diagram of the extended fractal draw experiment ... 113
Figure 45: Average timing results to finish the application ... 115
Figure 46: Time to send the components ... 116
Figure 47: Time to first draw ... 117

Figure 48: Time gained versus normal sending......119
Figure 49: Calculation of factorial (n) in Borg......134
Figure 50: Sequence diagram – classic strong migration of a factorial calculation......134
Figure 51: Sequence diagram - proactive migration of a factorial calculation......135
Figure 52: Watermark for delta definition in non-destructive memory......136
Figure 53: Possible pointers in non-destructive memory......137
Figure 54: Possible pointers and watermark in destructive memory......137
Figure 55: Borg calculation computational states......138
Figure 56: application without implicit stack operations......140
Figure 57: Application with a potential large delta......141
Figure 58: Large delta......141
Figure 59: Dealing with a large delta......142
Figure 60: Multi node hopping......153
Figure 61: Multi node hopping and evaluation......154
Figure 62: Proactive migration and evaluation......156

List of Tables

Table 1: Typical Migration Steps 26
Table 2: Average timing results (ms) for Benchmark application 78
Table 3: Standard deviation (ms) of timing results for Benchmark application 78
Table 4: Average timing results (ms) for CoolImage application 79
Table 5: Standard deviation (ms) of timing results for CoolImage application 79
Table 6: Average timing results (ms) for Gremlin application 81
Table 7: Standard deviation (ms) of timing results for Gremlin application 81
Table 8: Average timing results (ms) for adapted Gremlin application 82
Table 9: Standard deviation (ms) of timing results for adapted Gremlin application 82
Table 10: Properties of the Pre-fetching Technique 86
Table 11: Migration Steps Time Intervals 96
Table 12: Deployed Units 96
Table 13: Necessary Conditions for Removing Network Latency 99
Table 14: Average timing results (ms) to finish the application 115
Table 15: Standard deviation (ms) of timing results to finish the application 115
Table 16: Time to send the components (ms) 116
Table 17: Time to first draw (ms) 117
Table 18: Time gained versus normal sending 119
Table 19: Properties of the Component Stream Technique 126
Table 20: Stack and Dictionary Values during Evaluation Factorial Program 139
Table 21: Properties of the Proactive Migration Technique 144
Table 22: Summary of the performance indications 149

Samenvatting

Samenvatting in het Nederlands

In overeenstemming met het aanvullend facultair reglement doctoraat in de toegepaste wetenschappen

Titel

Progressieve Mobiliteit

Thesis

Progressieve Mobiliteit
Verbergt Netwerk Latentie

Ambient Intelligence (AmI) [ISTAG 2001] is de visie dat computer technologie onzichtbaar zal worden en zich zal integreren in de alledaagse voorwerpen rondom ons. De mensen zullen leven in een omgeving die zich bewust is van hun aanwezigheid, gevoelig is voor hun behoeften en hier ook op reageert [van Loenen 2003].

Om dergelijke intelligente omgevingen te bouwen zijn we genoodzaakt applicaties te implementeren in heterogene gedistribueerde netwerken, gekenmerkt door een vluchtige topologie als gevolg van de mobiele natuur van sommige van de knooppunten.

Energie, fouttolerantie en mobiliteit zijn de drie sleutelconcepten waar we mee zullen moeten omgaan om de ondersteuning te implementeren voor de toekomstige omgevingen. Specificatie, modelering en analyse van deze systemen zullen enkel mogelijk worden door gebruik te maken van een mobiliteitgebaseerde calculus [Lindwer et al. 2003] waar we de eigenschappen van communicatie zien als een dynamisch gegeven dat individuele knooppunten toelaat met elkaar te interageren, zich te verplaatsen en collectief een gegeven taak uit te voeren.

We staan voor de uitdaging om communicatiesystemen te ontwikkelen die de toekomstige omgevingen in staat stellen hun diensten aan te bieden en om applicatie scenario's te ontwerpen en implementeren om de intelligente omgeving tot leven te brengen.

"Het vinden van wegen om complexe taken te **partitioneren** en te distribueren, op een schaalbare wijze, tussen computationele knooppunten met beperkte hulpbronnen is mogelijk één van de meest uitdagende problemen die opgelost dienen te worden om computers naadloos te integreren naar omgevingen" [Lindwer et al. 2003].

Een valabele kandidaat om dynamische communicatiesystemen te implementeren tussen computationele knooppunten in een intelligente omgeving is mobiele code, code die over een netwerk verzonden kan worden en geëvalueerd[1] wordt op het platform van de ontvanger.

Een belangrijk probleem in verband met mobiele code is **netwerk latentie**, de tijdsvertraging als gevolg van het netwerk die optreedt voor de code geëvalueerd kan worden. Netwerk latentie wordt een kritische factor voor de bruikbaarheid van toepassingen die geladen worden over een netwerk, vóór de evaluatie van deze code kan starten. Vergeleken met andere tijdsvertragingen die zich voordoen als gevolg van code mobiliteit, is de tijd om een applicatie te transporteren van een zendend gastplatform naar een ontvangend gastplatform, veruit de meest tijdrovende activiteit, wat kan leiden tot betekenisvolle opstartvertraging van de toepassing. Dit is vooral het geval in netwerkomgevingen met een lage bit snelheid zoals sommige draadloze netwerken met groot bereik of in overbelaste netwerken.

Behalve netwerk latentie is er nog ander mogelijke tijdsvertraging als gevolg van het computersysteem zelf, zoals de (eventueel "juist op tijd") vertaling van de code en mogelijke evaluatie vertragingen als gevolg van het taakbeheer door het besturingssysteem of problemen met concurrente uitvoering. Deze vertragingen zullen we **systeem latentie** noemen.

In een mobiele omgeving wordt systeemperformantie uitgedrukt in **invocatie latentie**. Invocatie latentie is de tijdsvertraging tussen de invocatie van een toepassing en de start van de evaluatie van deze toepassing [Krintz et al. 1998]. Het is de combinatie van de netwerk latentie en de systeem latentie die zich voordoen voor de evaluatie van de toepassing start.

De vertraging tussen de invocatie van een toepassing en het verschijnen van de gebruikersinterface noemen we: gebruikersinterface latentie. Het is de som van de netwerk

[1] We gebruiken de meer algemene term evaluatie om de uitvoering of de interpretatie van code aan te geven.

latentie en de systeem latentie die zich voordoet voor het verschijnen van de gebruikersinterface.

Een ander potentieel probleem in tijdskritische toepassingen is de **toepassing beschikbaarheid**, het feit dat een toepassing tijdens zijn migratie niet beschikbaar is om wisselwerkingen aan te gaan met andere processen die daarop aansturen. In een klassiek migratie scenario zal een toepassing die migreert van een gastplatform naar een ander gastplatform altijd tijdelijk stilgelegd worden om pas terug opgestart te worden nadat de volledige code geladen is en hersteld in zijn oorspronkelijke vorm. De toepassing wordt pas terug beschikbaar na de volledige migratie fase.

Met de opkomst van intelligente omgevingen, gebouwd op een heterogeen gedistribueerd systeem, waar knooppunten het netwerk kunnen komen vervoegen of weggaan, zullen de connectietijden tussen de knooppunten onderling onvoorspelbaar worden. Een applicatie stoppen en pas terug opstarten nadat het vroeg of laat op zijn bestemming is gearriveerd zal deze mogelijk veel te lang onbeschikbaar maken voor de gebruiker.

Knooppunten in een intelligente omgeving kunnen onbereikbaar worden door de mobiliteit van de gebruiker, energiebeperkingen, plotseling optredende fouten enz…

In dergelijke netwerken, gekenmerkt door hun vluchtige topologie zal mobiele code een belangrijke rol spelen. Netwerk latentie en systeem latentie, de oorzaak van invocatie latentie zullen kritische factoren worden. Zo ook gebruikersinterface latentie, en toepassing beschikbaarheid.

Deze thesis onderzoekt **progressieve mobiliteit** en stelt verschillende migratie scenario's voor om bovenstaande problemen aan te pakken. De voorgestelde technieken partitioneren de toepassing in componenten en migreert deze componenten, progressief in de tijd, naar de ontvanger.

Het doel van ons onderzoek is om de haalbaarheid aan te tonen van enkele scenario's die het impliciete parallellisme aanwenden dat zelfs in de meest eenvoudige computernetwerken gevonden wordt maar massaal aanwezig zal zijn in intelligente omgevingen.

Dit onderzoeksdomein staat nog in de kinderschoenen en er is weinig ondersteuning van bestaande methodologieën en gereedschappen. Daarom is het ook niet onze intentie volledigheid of universaliteit na te streven.

Voor de huidige stand van zaken i.v.m. het uitwerken van deze ideeën verwijzen we naar [Krintz et al. 1998] waar onderzoek werd uitgevoerd door het vooraf sturen van methodes in een Java programmeeromgeving te simuleren.

De scenario's kunnen ingezet worden op het systeem niveau, op het niveau van lokale netwerken en in wereldwijde Internet netwerken hoewel de impact groter zal zijn in netwerken met een lage bit snelheid. De geëxploreerde thema's hebben als gemeenschappelijk kenmerk dat zij allen een progressief migratie scenario toepassen en worden genoemd:

- **Progressieve anticiperende mobiliteit met het op voorhand laden van gepermuteerde code**
- **Progressieve mobiliteit met component stromen**
- **Progressieve anticiperende mobiliteit met pro-actieve migratie**

Progressieve anticiperende mobiliteit met het op voorhand laden van gepermuteerde code is een techniek die de toepassingscode permuteert en parallellisme exploiteert tussen het

laden en evalueren van de code met het doel om de gebruikersinterface latentie te reduceren. De techniek is geïnspireerd op stromende (eng: *streaming*) media waarbij de evaluatie van de gegevens bij de ontvanger van start gaat lang voordat de volledige datastroom is binnen geladen.

De techniek laat toepassingen toe hun evaluatie vroeger te starten, vooral toepassingen die tijdens de opstart fase sterk voorspelbaar zijn, zoals toepassingen die starten met het bouwen van een standaard grafische gebruikersinterface, wat een grotere mate van anticipatie toelaat [Stoops et al. 2002].

Onze bijdragen met deze techniek zijn:

- Automatische permutatie van vertaalbare eenheden gebaseerd op een afhankelijkheidsgrafiek zodanig dat de statische structuur van de voorstelling van de applicatie in een bestandstructuur een reflectie is van zijn dynamisch gedrag
- Introductie van een synchronisatie mechanisme ter ondersteuning van de evaluatie van code die slechts partieel ter beschikking is
- Op voorhand laden van gepermuteerde code met focus op gebruikersinterface latentie

Experimenten in Smalltalk [Goldberg and Robson 1989] bevestigen ons thesis statement dat progressieve mobiliteit netwerk latentie verbergt en systeem latentie reduceert door aan te tonen dat deze techniek het mogelijk maakt netwerk latentie te verbergen door parallellisme toe te passen tussen het laden van de code en de evaluatie van de code. Daar de evaluatie van de toepassing vroeger start zal de invocatie latentie en dus ook de systeem latentie en vooral de gebruikersinterface latentie afnemen.

Als praktische validatie hebben we deze aanpak getest op drie toepassingen met elk een typisch doch verschillend gedrag.

Met onze experimenten bereiken we een gemiddelde gebruikersinterface latentie die 25% is van de originele gebruikersinterface latentie. Een reductie van de totale evaluatie tijd van de toepassing van 70% kon bereikt worden voor sommige toepassingen.

Progressieve mobiliteit met component stromen is een techniek waarbij componenten van een toepassing één voor één migreren van het ene gastplatform naar het andere totdat de volledige toepassing gemigreerd is. Deze techniek is ook geïnspireerd op stromende media waar de evaluatie van de gegevens start lang voordat de volledige stroom gegevens is geladen. Vergeleken met de vorige techniek, waarbij statische code wordt gemigreerd, zal deze techniek progressief draaiende toepassingen migreren zonder ze tijdelijk stop te zetten. Tijdens de migratie fase zal de toepassing beschikbaar blijven voor communicatie over en weer met andere processen. Wanneer een toepassing "stroomt" zal een gedeelte ervan reeds op het ontvangende platform draaien terwijl een ander deel nog op het zendend platform draait.

Onze bijdrage met deze techniek is:

- Introductie van progressieve mobiliteit met component stromen

Experimenten in Borg [Van Belle et al. 2001], Java en Smalltalk bevestigen ons thesis statement door aan te tonen dat het mogelijk is netwerk latentie volledig te verbergen. We tonen ook aan dat het mogelijk is de systeem latentie in het algemeen en gebruikersinterface latentie in het bijzonder te reduceren door de optimale migratie sequentie te kiezen voor de verschillende componenten. Door een strategie met parallel processing toe te passen is het mogelijk de totale evaluatie tijd van de toepassing te reduceren en daardoor ook de systeem latentie.

We tonen de uitvoerbaarheid van het concept aan door middel van een experiment in Borg en breiden dit achteraf uit door de techniek van progressieve mobiliteit met component stromen te implementeren op een toepassing die een fractal tekent onder controle van een grafische gebruikersinterface in Java en nadien implementeren we deze toepassing met verschillende concurrentie strategieën in Smalltalk [Devalez 2003]. In al deze omgevingen verkregen we betekenisvolle resultaten.

We bieden ook richtlijnen aan om toepassingen op design niveau te optimaliseren met het oog op maximalisatie van het rendement van de techniek.

Progressieve anticiperende mobiliteit met pro-actieve migratie is een techniek die begint met het nemen van een snapshot van de volledige toepassing vóór de echte migratie van start gaat. Deze snapshot bevat ook de volledige evaluatie toestand van de toepassing. Dan wordt deze snapshot anticipatief gemigreerd naar een potentiële ontvang gastplatform terwijl de oorspronkelijke toepassing verder wordt geëvalueerd. Wanneer de echte migratie, naar datzelfde gastplatform getriggerd wordt, dient er enkel nog het verschil tussen de huidige evaluatie toestand van de toepassing en deze die al bevat zat in het snapshot gemigreerd te worden. Dit verschil zal veel kleiner zijn dan de originele code waardoor de migratie tijd betekenisvol gereduceerd kan worden.

Onze bijdragen met deze techniek zijn:

- Introductie van Progressieve anticiperende mobiliteit met pro-actieve migratie

Vermits de techniek sterk afhankelijk is van reflectie en reïficatie van de datastructuren die de evaluatie toestand bevatten, implementeerden we een prototype in Borg om de haalbaarheid aan te tonen. Borg is een programmeeromgeving ontwikkeld aan ons labo, waardoor de nodige kennis over typische implementatie kenmerken aanwezig is.

We zullen concluderen dat progressieve mobiliteit de performantie van een mobiele toepassing verhoogt. De voorgestelde technieken kunnen de *totale evaluatie tijd* verminderen, *netwerk latentie* verbergen, *invocatie- en gebruikersinterface latentie* reduceren en de *beschikbaarheid* van toepassingen verbeteren.

Summary

Ambient Intelligence (AmI) [ISTAG 2001] is the vision that computer technology will become invisible, integrated in all the everyday objects around us. People will live in an environment that is aware of their presence, and is sensitive and responsive to their needs [van Loenen 2003].

In order to build these intelligent ambients we are challenged to implement applications in a heterogeneous distributed network characterized by a volatile topology due the mobile nature of some of the nodes in the network.

Energy, fault-tolerance and mobility are the three key concepts that we will have to deal with to implement the support for the future ambients. Specification, modeling and analysis of such systems will become possible only by if we deal with communication properties, as a dynamic phenomenon that enables individual nodes in the network to interact, move around and collectively perform a given task.

We are faced with the challenge to develop communication systems that empower the future environments and the design and implementation of application scenarios that bring ambient intelligence to life.

“Finding ways to **partition** and distribute complex tasks, in a scalable manner, among computational nodes with limited resources represents perhaps one of the most challenging problems that need to be solved in order to seamlessly integrate computers into ambients” [Lindwer et al. 2003].

A valid candidate to implement dynamic communicating systems between computational nodes in the ambient network is mobile code: code that can be transmitted across the network and evaluated[2] on the receiver's platform.

An important problem related to mobile code is **network latency:** the time delay introduced by the network before the code can be evaluated. Network latency becomes a critical factor in the usability of applications that are loaded over a network before their evaluation can start. Compared to other time delays involved with code mobility, the transfer of the code from the sending host to the receiving host is in general the most time-consuming activity, and can lead to significant delays in the startup of the application. This is especially true in the case of low data rate environments such as the some of the current wireless, wide area, communication systems or in overloaded networks.

Besides network latency there are other possible delays introduced by the computer system, e.g. (Just In Time) compilation, evaluation delay caused by the (distributed) operating system scheduling or concurrency problems. These delays will be called **system latency**.

In a mobile environment system performance is measured by **invocation latency**. Invocation latency is the time from application invocation to when the evaluation of the program actually begins [Krintz et al. 1998]. It is the combination of the network latency and the system latency that occurs before the evaluation of the application starts.

The delay between the invocation of an application and the appearance of the user interface will be called: **user interface latency**. It is the sum of the network latency and the system latency that occurs before the appearance of the user interface.

Another potential problem in time-critical applications is **application availability,** the fact that an application will not be available during its migration for other processes that need to interact with it. In a classical migration scheme the application that migrates from one host to

[2] We utilize the more general term evaluation to describe the execution or interpretation of the code.

another is temporarily halted and is restarted at the receiving host after the code is completely loaded and restored in its original form. It will become available again only after the complete migration phase.

In the advent of ambient intelligence environments built on a heterogeneous distributed system, where nodes may join or leave the network, connection times between the nodes become unpredictable. Therefore, also the time needed to migrate the application will become unpredictable and just halting the application until it may arrive at its destination, sooner or later, may not be an option any more.

Ambient intelligence network nodes may become unreachable due to the mobility of the user, energy source constraints, intermittent failures etc... In such networks, characterized by their volatile topology, mobile code will play a major role. Network latency and system latency, the cause of invocation latency and user interface latency, together with application availability will become critical factors.

This dissertation investigates **progressive mobility** and explores different migration schemes to cope with these problems. The explored themes all have in common that we partition the mobile code and migrate the parts, **progressively** in time, to the receiver.

Our research goal is to provide a proof of concept of scenarios to harness the implicit parallelism, found in even the most simple computer networks, but that will be massively available in ambient intelligence environments. Given that this research area is still in its infancy, without much support from existing methodologies and tools it is not our intention to pursue completeness or universality.

As previous work in developing these ideas we refer to [Krintz et al. 1998] where, among other things, research on non-strict evaluation was conducted by running simulations of method pre-fetching in a Java programming environment.

The scenarios can be deployed at the system level, in local area networks and in worldwide internet networks although the impact will be more significant in low data rate networks. The explored themes all have in common that they apply a progressive migration scheme, and are called:

- **progressive anticipative mobility using pre-fetching of permuted code**
- **progressive mobility using component streams**
- **progressive anticipative mobility using proactive migration**

Progressive anticipative mobility using pre-fetching of permuted code applies a technique that permutes the application code and exploits parallelism between loading and evaluation of code to reduce user interface latency. The technique is inspired by streaming media where the evaluation of the data at the receiver starts long before the complete data stream is loaded.

The technique allows applications to start their evaluation early, especially applications with a predictable startup phase, such as building a standard GUI, which allows a higher level of anticipation [Stoops et al. 2002].

Our contributions with this technique are:

- Automatic permutation of compilable units based on a dependency graph so that the static structure of the representation of the application in a file structure reflects its dynamic behavior
- Introduction of a synchronization mechanism to support evaluation of code that is only partially present

- Pre-fetching of permuted code with focus on user interface latency

Experiments in Smalltalk [Goldberg and Robson 1989] confirm our thesis statement that progressive mobility hides network latency. We show that this technique is able to hide network latency by applying parallelism between the code loading and the evaluation of the code. Since the evaluation of the applications will start early, invocation latency and therefore also system latency and especially user interface latency will decrease.

As a practical validation we tested our approach on three applications each exhibiting some typical but distinct behavior. In our experiments we obtain an average user interface latency that is 25% of the original user interface latency. For one of the applications the total evaluation time could be reduced to 70% of the original evaluation time.

Progressive mobility using component streams is a technique where components of an application migrate one by one from one host to another until the application is completely migrated. This technique is also inspired by streaming media where the evaluation of the data at the receiver starts long before the complete data stream is loaded. In contrast with the previous technique, where static code is migrated, this technique progressively migrates running applications without halting them. During the migration phase the application will remain available for interaction with other processes. When streaming a running application, part of the application will already run on the receiving host while the other part is still running on the sending host.

Our contribution with this technique is:

- Introduction of progressive mobility using component streams

Experiments in Borg [Van Belle et al. 2001], Java and Smalltalk confirm our thesis statement by showing that is possible to hide network latency completely. We also show that it is possible to reduce system latency in general and user interface latency in particular by choosing the optimal sequence of migration of the different components. By applying a strategy with parallel processing we are able to decrease the total evaluation time of the application and therefore the system latency.

We perform a proof of concept experiment in Borg and elaborate on this by implementing the technique of progressive mobility using component streams on a fractal drawing application with a graphical user interface in Java. Afterwards we will deploy the fractal drawing application with different concurrency strategies in Smalltalk [Devalez 2003]. In all these environments we are able to achieve significant results.

We also provide guidelines to optimize applications at the design level to increase their profit from the technique.

Progressive, anticipative mobility using proactive migration is a technique that starts by taking a snapshot of the entire application before the real migration is triggered. This snapshot includes the applications computational state. Then this snapshot is anticipatively migrated to a potential receiving host while the original application continues to run. Then, when the real migration, to that same receiving host, is triggered, there is only the need to send the difference of the current computational state and the state that was already contained in the snapshot. This difference will be significantly smaller than the original code which may reduce the migration time considerably.

Our contributions with this technique are:

- Introduction of progressive, anticipative mobility using proactive migration

Since the technique is heavily dependent on reflection and reification of the computational state we implement a proof of concept in Borg, a programming environment developed at our lab which provides us with all the necessary knowledge on the idiosyncrasies of the implementation of the environment.

We will conclude that progressive mobility may increase the performance of a mobile application. The proposed techniques can enhance *overall program evaluation time*, hide *network latency* reduce *invocation latency*, *user interface latency*, and improve *application availability*.

1 Introduction

1.1 Thesis Motivation

With the advent of *Ambient Intelligence* (AmI) [ISTAG 2001], where people will be surrounded by intelligent and intuitive interfaces embedded in everyday objects around us, mobile code will become an important medium to support this intelligent environment. In particular everyday objects such as telephones, watches, lights, doorbells should be able to communicate with one another and with the users through wireless networks. Objects that do not move relatively with respect to each other can rely upon current communication protocols to provide a stable connection but the connection between moving objects: a car, a bicycle, an occupant walking around in his house poses new challenges.

Ambient intelligence, as defined by the EC Information Society Technologies Advisory Group builds on three recent key technologies: *Ubiquitous Computing, Ubiquitous Communication and Intelligent User Interfaces*. Ubiquitous computing means integration of microprocessors into everyday objects like furniture, clothing, toys, even paint that cleans off dust and notifies you of intruders, walls that selectively dampen sounds [Weiser and Brown 1996]. Ubiquitous communication enables these objects to communicate with each other and the user by means of ad-hoc and wireless networking. An Intelligent user interface enables the inhabitants of the AmI environment to control and interact with the environment in a natural (voice, gestures) and personalized way (preferences, context) [ISTAG2001].

Ambient systems need to address some key issues [O'Hare et al. 2004]:

- Recognition and accommodation of the diversity of devices that contribute to the organic nature of the ambient and ubiquitous computing nervous system
- The need for personalisation and system adaptivity
- An understanding of the dynamics of context
- Provision of support for collaboration and cooperation between distributed ambient system components
- Delivery of systems that exhibit autonomic characteristics yielding self management and self healing capabilities

In addressing these core issues, we see that developers more and more adopt an agent-based approach [O'Hare et al. 2004]. Information between two objects can be exchanged by just sending data between them but to address the issues mentioned her above we need to send behavior (code) also. In terms of functionality one can imagine a personal agent that follows a user in time and space by migrating to objects that are in the neighborhood of the user so that its service is easily accessible. This thesis will only focus on the exchange of code.

In order to support this new network architecture where connections between partners are no longer predictable and where the connection time may be less than a second there is a new need for techniques to exchange information as fast as possible. These new techniques should make optimal use of this type of unpredictable, unstable and time constrained connections. Just sending the information and hoping that it will arrive sometimes and that it still will be usable once it gets there, as current protocols do, will no longer be an option.

If a block of data needs to be sent to a moving target, and one cannot predict the width of the current timeframe, one possible solution is to break up the block of data in smaller parts and send them one by one to the receiver. This will increase the possibility that they will fit in the temporal timeframe. Precaution should be taken to send the most important parts first, in a format that makes these partial data immediately usable at the receiver’s end.

Mobile code is a plausible candidate to ensure the connection between different moving software components or devices. The emerging technique of mobile code is a new promising way to set up communication mechanisms between different parties but there is still much research needed to develop techniques to support and optimize these communication mechanisms.

“Mobility challenges old assumptions and demands novel software engineering solutions. Coordination mechanisms must be developed to bridge effectively a clean abstract model of mobility and the technical opportunities and complexities of wireless technology, device miniaturization, and code mobility. Logical mobility opens up a broad range of new design opportunities, physical mobility forces consideration of an entirely new set of technical constraints, and the integration of the two is an important juncture in the evolution of software engineering as a field” [Gruia-Catalin 2000] .

Since the width of the timeframe available to migrate the code is not predictable, we need some kind of mechanism to break up code into smaller parts and send them one by one, progressively in time, to the receiver. This will increase the possibility that they will fit in the temporal time frame. Here too, precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately usable (ready for evaluation) at the receiver’s end. Since connections between hosts in these new environments are more volatile than in static networks there is also the need for mechanisms that allows the code to continue its evaluation during the progressive migration so that the application remains available for users or other applications at all time.

Partitioning code and send the most important parts first together with application availability are the two main sub-problems in the domain of mobile code that this thesis focuses on. As a significant bonus, we will have our migrated code up and running much faster than by using traditional migration techniques. This makes the technique of progressive mobility, the division of the application in components and migration of these components, progressive in time, to the receiver, also applicable in current stable networks to speedup the start of an application that is loaded over a network and even for an application that is loaded from a hard disk.

1.2 Latency

Research has shown that invocation latency, the time from application invocation to when the evaluation of the program actually starts is crucial in how users view the performance of an application. Early work investigating the effect of time-sharing systems [Doherty and Kelisky 1979] where different users run their applications on the same machine concluded that increased system response time disrupted user thought processes. More recent work [Johnson 98] investigates the impact of unpredictable web latency, the time between the click on a hyperlink and the appearance of the result.

Invocation latency of an application residing on a single computer is greatly reduced nowadays by the introduction of Gigahertz microprocessors and fast internal bus systems, but it still remains significant in a mobile code environment, a software system distributed over a physical or logical network of heterogeneous computers.

An important problem related to mobile code is **network latency**: in this context we define network latency as the time delay introduced by moving the code over the network before it can be evaluated. This delay typical has several possible causes (Table 1). The code must typically be (1) halted, (2) packed (3) possibly transformed in a compressed and/or secure

format, (4) transported over a network to the target platform, (5) possibly retransformed from its compression or security standard, (6) checked for errors and/or security constraints, (7) unpacked, (8) possibly adapted to the receiving host by compiling the byte codes or some other intermediate representation and finally (9) resumed.

Table 1: Typical Migration Steps

Step	Action
1	Halt the application
2	Pack it
3	Transform it
4	Transport to the receiver
5	Retransform it
6	Check it
7	Unpack it
8	Adapt it
9	Resume the application

The fourth phase, the transportation over the network, is in general the most time consuming activity, and can lead to significant delays in the startup of the application. This is especially the case in low data rate environments such as the current wireless communication systems or in overloaded networks. Therefore, it is imperative that in order to reduce the network latency we need to tackle this transportation phase.

Network latency becomes a critical factor in the usability of applications that are transported and eventually compiled "Just In Time". In a mobile environment application performance is measured by invocation latency. *Invocation latency* is the time from application invocation to when the evaluation of the program actually begins [Krintz et al. 1998]. Invocation latency is crucial in the user’s perception of the performance of an application.

Almost 70% of all delay (transfer or compilation) of an average Java program occurs in the first 10% of program execution [Krintz 2001]. This result suggests that if the first 10% of an application is predictable then the delays can be reduced considerably. We found that in the applications we used for our experiments that the predictable zone ranged from 15 % to 30%.

In ambient intelligence environments the speed of migration might become a critical factor if an application migrates by example over a Bluetooth connection to a moving PDA. Bluetooth is a specification for short-range wireless connections with a maximum range of 10 meters. In this case the moving target might be only a few seconds in range, so there will be no time to waste and some worst case scenarios it will be only possible to send parts, subcomponents of the complete application.

1.3 Application availability

Application availability is a second potential problem. In time-critical applications it may not be acceptable that an application will not be available for other processes that need to interact with it during its migration, in short, the application must keep running at all times. If a classic migration scheme is deployed the application will become available again only after the complete migration phase which may take too much time. In a control engineering environment for example the maximum time between the intakes of samples of the quantity under control is strictly defined and if the sample timing exceeds this threshold just once this

may compromise the complete control process. A typical approach to provide high availability is replication of data and services but this is an expensive solution [Ladin et al. 1992].

The potential problem of application availability is expected to increase in ambient intelligence environments where the transport time for an application can be limited to a few seconds or a fraction of a second, i.e. the time a moving client is connected to a sensing point of its surrounding intelligent environment. These limitations in time combined with low data rates of mobile devices can force us to split up the running application dynamically in different parts so that, during each connection, part of the application could be transferred. During this transfer the evaluation of the different parts of the application continues.

1.4 Research Goals

We need to gain more insight in the different forms of latency and application availability of applications that migrate over different kind of networks. Network latency is one of the most important potential problems in mobile code environments, therefore we like to investigate in different kind of settings in order to reduce or hide this network latency. We will do this with a special focus on user interface latency since this kind of latency will determine the users perception of the performance of a mobile application.

Our main research goal is to investigate in how to **harness the implicit parallelism** that is found in computer networks. Even for the most simple network connection between two computers there are at least two processors available. Modern computer architectures also provide a separate processor to manage the network traffic and we anticipate on upcoming new technologies where we may expect up to four different independent processors on one chip. As a side effect of the introduced parallelism, we may also expect to see some reduction in system latency.

To overcome invocation latency we can send the code in compressed form or increase the data rate of the transport channel. This is not generally applicable however to the wide variety of code formats and transport channels that are available for mobile code where the maximum data rates are dictated by the underlying physical levels of the network.

In order to break open this new, complex and difficult research domain we **explore three different themes** in which we use different perspectives to take advantage of the implicit parallelism.

It is our goal to provide a proof of concept of the different scenarios developed under these themes without pursuing completeness or universality. The three themes relate to the perspective of:

- Pre-fetching of permuted code
- Deploying component streams
- Proactive migration

A common characteristic of the investigated scenarios in these themes is that we split up the code and send the parts one by one **progressive in time** over the network. This is the essence of progressive mobility.

The first theme, *progressive anticipative mobility using pre-fetching of permuted code,* is tailored to applications that have not started yet before they migrate.

We apply a technique that automatically permutes the application at the level of compilable units based on a dependency graph and exploits parallelism between transportation and

evaluation of code to hide network latency. We perform pre-fetching experiments to hide network latency in the Smalltalk programming environment.

The technique allows many applications to start evaluation earlier on, especially applications with a predictable, deterministic start-up phase (such as building a GUI) [Stoops et al. 2002]. A synchronization mechanism guards the availability of the units that are invoked during application evaluation. Our contributions on this theme are:

- Automatic permutation of compilable units based on a dependency graph so that the static structure of the representation of the application in a file structure reflects its dynamic behavior
- Introduction of a synchronization mechanism to support evaluation of code that is only partially present
- Pre-fetching permuted code with a focus on user interface latency

On the first theme we migrate static code, code that is not evaluated at the time the migration is triggered, but sometimes we need to migrate running applications as well. Think of an active agent application that needs to follow its user in an ambient intelligent environment. This becomes our second theme. In order to split up the code of an application that is already running, we introduced: *progressive mobility using component streams*.

A running application can migrate to another host by migrating its components one by one. Preferably we will migrate components in a time slot when they are not needed for evaluation. In this case the evaluation of the application continues during its migration which results in a high application availability.

We perform experiments to hide network latency, to reduce system latency and to exploit parallel evaluation. We deliver a proof of concept in the programming environments of Borg, Java and Smalltalk. We show that network latency can be hidden almost completely, that system latency can be reduced and that parallel evaluation on the sender and receiver can enhance the users perception of the migrating application.

Our contribution on this second theme is:

- Introduction of progressive mobility using component streams

In some cases it makes sense to migrate a running application proactively to a candidate receiving host, this is the third theme we investigated into. We provide a proof of concept in the Borg programming environment, an environment that provides the reflection we need to explore this theme.

On this last theme our contribution is:

- Introduction of progressive, anticipative mobility using proactive migration

1.5 Research restrictions

We do realize that we enter a new unexplored research domain, so besides the restrictions imposed by current practical networks and in order to make it realistic to obtain useful results we adopt certain restrictions in our experimental environment.

First of all, for most of our experiments, we relied on stable fixed TCP/IP network connections. Conducting the experiments in unstable, unpredictable or vulnerable networks would lead us too far away from our research goal.

We did not investigate in connection-oriented networks as GSM networks and we neglected possible security aspects. Splitting up code in smaller parts may introduce extra security risks so that possibly an authification for each part might be appropriate but this problem was not considered in the context of this research.

The hosts used in our experiments have always comparable processing power and we only investigate the progressive migration of code. Possible simultaneous progressive migration of data is left for future work.

In all our experiments, we applied a push strategy. In our exploration we assume that the know-how and know-when of the migration of partitioned code is located in the sending host. However, this does not exclude the possibility of successful combinations with a pull-strategy and we will mention these opportunities but we did not implement it in our experiments.

Besides these restrictions imposed by ourselves we are also confronted with limitation of the programming languages or the tools used.

We find us confronted with the lack of support for strong mobility in popular programming environments as Java and Smalltalk and a relative high class-level granularity of code blocks in Java.

For most of the problems we implemented a workaround. We applied the µCode toolkit [Picco 1998] to introduce a limited form of strong mobility in Java. With this toolkit, we where able to migrate threads from one host to an other inclusive the state of its variables. In contrary with real strong mobility the runtime stack is not migrated. Therefore, the threads needed to get started again on the receiving host.

The implementation of a communication system between threads in Java that operate in different namespaces is also far from trivial. As a workaround we have setup a dedicated communication object accessible via remote method invocation (RMI).

In the Smalltalk environment we encountered several flaws in the current implementation of the VisualWorks programming environment. We needed to combine the standard Smalltalk Opentalk tool for distributed systems with the “Binary Object Streaming Service” BOSS to implement strong mobility for small processes.

1.6 Chapter Summaries

Chapter 2: A Conceptual Framework for Progressive Mobility

We start by presenting a conceptual framework to provide a context to describe the techniques of progressive mobility. The framework is limited to all the concepts that are needed as a base to describe the techniques and contains also related work in these domains.

We present a three dimensional conceptual framework for the mutually orthogonal dimensions: *Network, Application and Techniques*.

In exploring the network dimension, we will discover the boundaries that will indicate the window of opportunity for progressive mobility.

The application dimension describes relevant application properties as structure, size, user interfaces and components. In addition, the programming environments employed in this dissertation are presented here.

Finally, in the techniques dimension we explore relevant techniques as mobility, parallelism, pre-fetching and other related techniques.

Chapter 3: Progressive Anticipative Mobility using Pre-fetching of Permuted Code

In this chapter, we propose a technique that applies the idea of progressive transmission to software code. The evaluation of the digital code stream can start before the transportation phase is completed by anticipating on the sequence of evaluation.

We start by describing the technique of code permutation that will be employed to pre-fetch static code in order to speedup a weak mobility migration scheme. A technique to permute Smalltalk source code is presented. The Smalltalk source code is permuted at the level of compilable units in such a way that the units that are needed first during evaluation are placed at the start of the source file.

A prototype tool in Smalltalk is presented that automatically permutes a Smalltalk source file and generates a set of source code files optimized for the pre-fetching process.

The feasibility of the technique has been validated by implementing prototype tools in Smalltalk. In this chapter we will restrain ourselves to weak mobility. We will handle progressively strong mobility in chapter 4 and 5.

Chapter 4: Progressive Mobility using Component Streams

This chapter introduces the technique of progressive mobility with strong mobility instead of weak mobility. Progressive mobility using component streams allows applications to migrate from host to host without sacrificing evaluation time during the migration phase and it allows the application to start at the receiving host much earlier.

The technique is inspired by streaming media. When streaming a running application, part of the application will already run on the receiving host while another part is still running on the sending host.

The feasibility of the technique has been validated by implementing prototype tools in the Borg mobile agent environment and later also in Java and Smalltalk. Our experiments show that this migration strategy can hide network latency almost completely.

Chapter 5: Progressive Anticipative Mobility using Proactive Migration

This next chapter introduces yet another technique of progressive mobility using proactive migration, a technique that avoids the delays introduced by the former technique.

The technique sends the application code, including the computational state, in advance, anticipatively to the remote host, before the actual migration is requested. Then, when the actual migration takes place, we won't transfer the complete application but only the delta of the current computational state with the already migrated computational state. At the receiver the computational state can be brought up to date by applying this delta to the previous received computational state before evaluation is continued.

Chapter 6: Conclusion

We conclude in this chapter by summarizing the results obtained during the exploration of the three themes and present some future Work.

As future research directions, we suggest, besides other things, extensions on progressive anticipative mobility using pre-fetching of permuted code and architectural transformations to make applications streamable.

Progressive anticipative mobility using proactive migration and evaluation is also proposed as a new progressive migration scheme and suggestions are made to aggregate crosscutting concerns of progressive mobility in aspects in order to make use of aspect oriented software design, an upcoming software engineering technique.

2 A Conceptual Framework for Progressive Mobility

We present a three dimensional conceptual framework for the mutually orthogonal dimensions: **Network**, **Application** and **Techniques.** These dimensions play a major role in our research domain. In ambient intelligence environments with ubiquitous communication, different kind of objects will cooperate in a *network* architecture where *applications* will migrate from one node to the other. In order to facilitate these migrations we will need to apply new and current *techniques*.

The framework provides a context to describe the technique of progressive mobility. The framework provides the terminology, definitions and properties of the relevant entities in which we describe *Progressive Mobility*. The framework also includes references to related work.

Roadmap:

- **Network**
    - Architecture
    - Packet Switching
    - Data Rate
    - Delays in Computer Networks
    - Performance
    - Window of Opportunity
- **Application**
    - Internal Structure
    - Size
    - Granularity
    - Evaluation Time
    - Delay
    - User Interfaces
    - Predictability
    - Components
    - Distributed Systems and Applications
    - Choosing an Experimental Programming Environment
    - Programming Languages Used
        - Borg
        - Java
        - Smalltalk
- **Techniques**
    - Mobility
        - Host Mobility
        - Evaluator Mobility
        - Data Mobility
        - Code Mobility
        - Process Mobility
        - Weak and Strong Mobility
    - Server - Push versus Client - Pull
    - Parallelism
    - Reflection
    - Compression
    - Reordering and Pre-fetching
    - Progressive techniques
    - Other related techniques

2.1 Network

In this section we describe the relevant entities of the network dimension of the conceptual framework.

2.1.1 Architecture

Network software is highly structured. Most networks are organized as a stack of layers, each one built upon the one below. The set of layers and protocols is called a network architecture. Two important network architectures are the OSI reference model and the TCP/IP reference model [Tanenbaum 2003]. Figure 1 shows the different names for the layers.

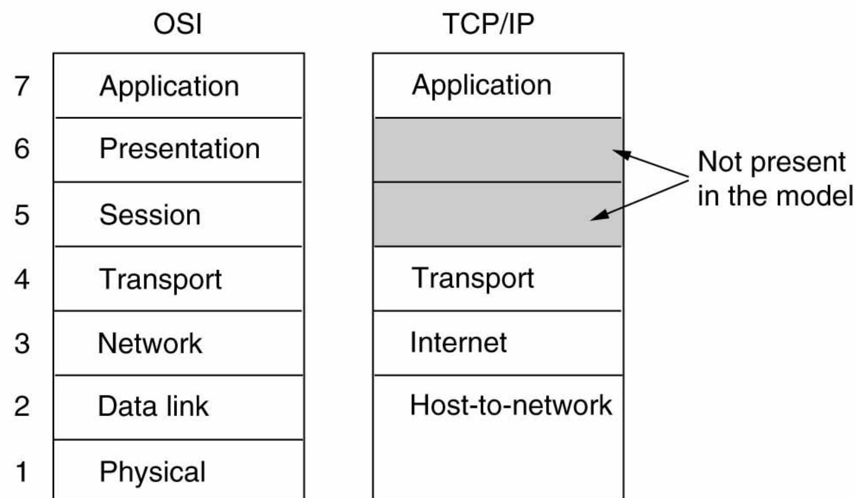

Figure 1: OSI and TCP/IP reference model

Many of the current networks use the internet transport protocol in the transport layer. The task of the transport layer is to provide reliable, cost-effective data transport from the sending host to the receiving host, independently of the host-to-network layer. Be it Ethernet on twisted pair or wireless LAN (802.11), Bluetooth, GSM, GPRS or UMTS to mention a few well known implementations of the lower layers.

2.1.2 Packet Switching

When a sending host has a message to send to a receiving host, the sending host first cuts up the message into packets, each one bearing its number in the sequence. These packets are then injected into the network one at a time in quick succession. The packets are transported individually over the network. These packets may follow different routes and may arrive in a different order at the end point. Finally they are disposed at the receiving host, where they are reassembled into the original message. A stream of packets is show in Figure 2 [Tanenbaum 2003].

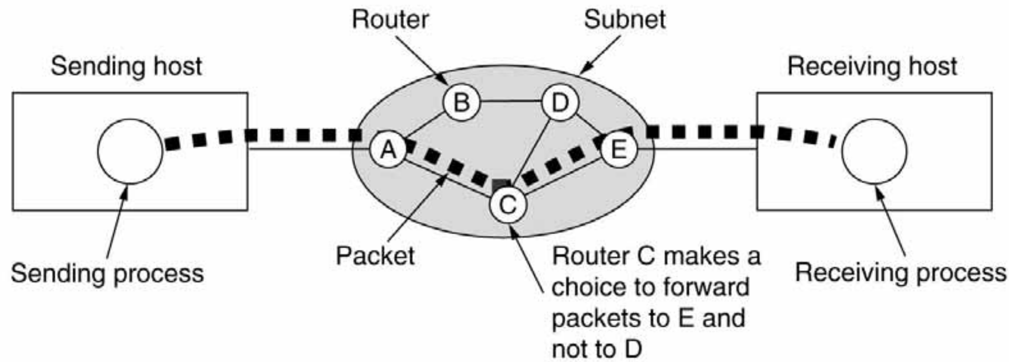

Figure 2: A stream of packets

If the message is not cut up into packets (message switching) there is no limit on data size, which means that routers must have disks to buffer long blocks of data. It also means that a single block can tie up a router line for a long time thereby preventing other traffic. Another advantage of packet switching is that the first packet of a message can be forwarded while the next packets have not fully arrived yet, reducing delay and improving throughput. For these reasons, computer networks are usually packet switched.

2.1.3 Data Rate

The speed of wireless data communications has increased enormously over the last years with the emergence of technologies as HSCSD (High Speed Circuit Switched Data), GPRS (General Packet Radio Services), and UMTS (Universal Mobile Telecommunications Service)

HSCSD is evolved from circuit switched data within the GSM environment. HSCSD enables the transmission of data over a GSM link at speeds of up to 57.6 kbps[3]. This is broadly equivalent to providing the same transmission rate as that available over one ISDN B-Channel.

GPRS is a packet-based wireless communication service that delivers data rates from 56 up to 114 Kbps and continuous connection to the Internet for mobile phone and computer users.

UMTS (Universal Mobile Telecommunications Service) is a third-generation (3G) broadband, packet-based transmission of text, digitized voice, video, and multimedia at data rates up to 2 Mbps. The service offers a consistent set of services to mobile computer and phone users no matter where they are located in the world and is based on the Global System for Mobile communication standard (GSM). [4]

2.1.4 Delays in Computer Networks

To emphasize that delay in this context mostly means wasted time, the term **latency** is often used. Latency was originally defined as**: "**the length of time it takes to respond to an event" [Barbacci 95]. Network latency however is now an expression of how much time it takes for a packet to get from one designated point to another.

---

[3] One kbps equals 1000 bits per second. - To avoid ambiguity, in this dissertation, we use the SI and IEC prefixes. [IEEE 1997] [IEC 2000] - see http://physics.nist.gov/cuu/Units/prefixes.html.
[4] A comparison of different communication speeds can be found at http://www.hawaii.edu/infotech/speeds.html and http://whatis.techtarget.com/definition/0,,sid9_gci214198,00.html [Aug 2003].

The latency assumption seems to be that data should be transmitted instantly between one point and another (that is, with no delay at all). The contributors to network latency include:

Propagation: This is simply the time it takes for a packet to travel directly from one place and another at the speed of light.

Transmission (transport): The medium itself (whether optical fiber, wireless, or some other) introduces some delay. The size of the packet introduces delay in a trip since a larger packet will take longer to receive than a short one.

Router and other processing: Each gateway node takes time to examine and possibly change the header in a packet (for example, changing the hop count in the time-to-live field).

Other computer and storage delays: Within networks at each end of the journey, a packet may be subject to storage and hard disk access delays at intermediate devices such as switches and bridges.

2.1.4.1 Delays in Connection-oriented Networks

Circuit switched networks as GSM need some extra connection time to establish a connection and after the connection is used the connection should be released. During the connection the sender just pushes bytes in at one end and the receiver takes them out at the other end.

When a connection is established there will be sometimes a negotiation between the sender, the receiver and the network about parameters to be used, such as data rate, maximum message size, quality of service and other issues. Typically, one side makes a proposal and the other side can accept it, reject it, or make a counterproposal [Tanenbaum 2003].

These connection setup times can play a significant role if one considers implementing progressive mobility where basically one message is split up in several smaller messages which may lead to the introduction of several connection setup times. We will discuss this aspect for each of the explored themes.

2.1.4.2 Delays in Connectionless Networks

Packet switched networks as GPRS and TCP/IP don't need the extra connection time as needed in connection-oriented services, each packet carries the full destination address and each packet is routed trough the system independent of the others. Normally, when two packets are sent to the same destination, the first one sent will be the first one to arrive. However if the first one is delayed it is possible that the second one arrives first. It is the responsibility of the receiving host to restore their original order.

Splitting a message in packets and reassembling them takes time and the larger the message is to send the more overhead time we may expect. In a classical TCP/IP protocol the packet size ranges from 1 to 2 kByte. Sending a message of 1000 bytes instead of 500 bytes does not increase the number of packets (only one in this case) and will not lead to a significant extra delay.

Figure 3 shows the time needed to transport an array of Unicode characters with sizes ranging from 1 element (16 bits) to 10^7 elements (1.6 E+08 bits) between two hosts[5].

[5] The experiment was carried out between two Dell® Inspiron 8100 computers with Intel® Pentium® III Mobile CPU AT/AT compatible processor at 1GHz processor speed and 256 Mb RAM running Windows® 2000 and VisualWorks 5i4.

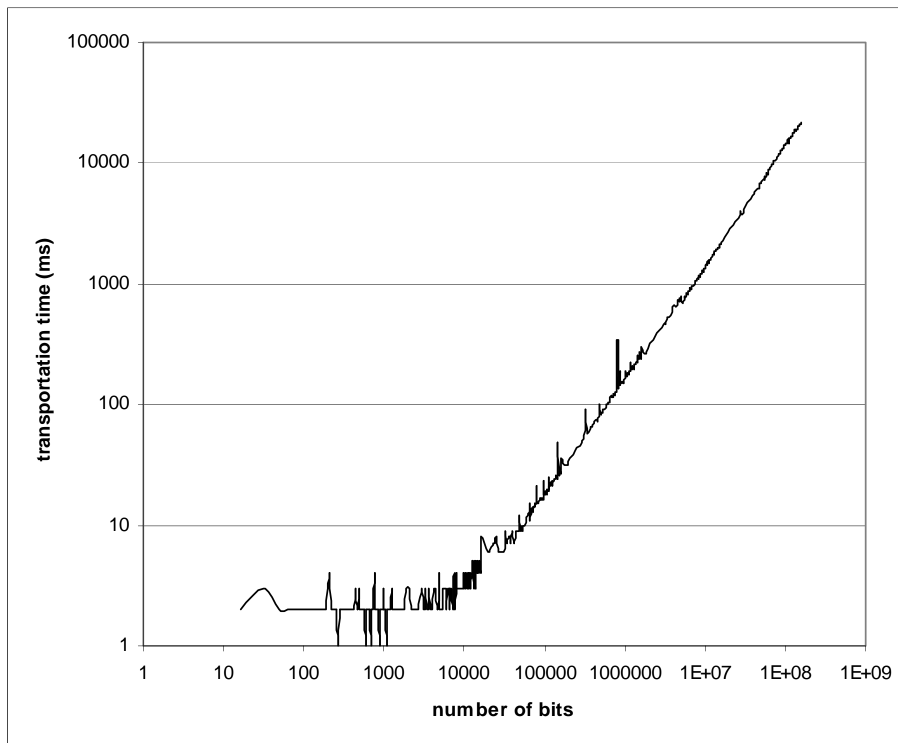

Figure 3: Time needed to transport character arrays of different sizes in a TCP/IP network

The figure shows that the transportation time becomes linear with the size of the message once the size of the message exceeds the packet size i.e. 2 kByte (16384 bits). Note that the graph uses logarithmic scales for both axes. The noise on the graph is mainly caused by garbage collection actions in the Smalltalk environment. The maximum resolution of the time measurement in the Smalltalk environment is 1 ms. For small arrays the transportation time is about 2ms with some variations from 1 ms to 4 ms, this explains the up and down "glitches" in the range from 10 to 1000 bits.

2.1.5 Performance

The current fiber technology is able to achieve data rates in excess of 50,000 Gbps (50 Tbps). The current practical limit of about 10 Gbps is due to our inability to convert between electrical and optical signals any faster [Tanenbaum 2003]. In the race for maximum speed between computing and communication in laboratory environments, communication won.

The TCP/IP protocol is also responsible, at the transport layer, to provide a reliable connection between a sending host and the receiving host. This means that the arrival of packets must be confirmed by the receiver by sending an acknowledge packet to the sender. If after a certain time the sender did not receive an acknowledgment for a packet with a particular sequence number, that packet will be sent again. If too many packets get lost, the sender will assume a network congestion and will slow down in order to try to pass al its packets.

In Figure 4 we show the time it takes to transfer a 1 Megabit file 4000 km over a fiber connection at various transmission speeds. Note that both axes have a logarithmic scale. Up to 1 Mbps the transport time is dominated by the rate the bits can be send. By 1 Gbps, the 40 ms round-trip delay dominates the 1 ms it takes to put the bits on the fiber. Further increases in data rate have hardly any effect at all [Tanenbaum 2003].

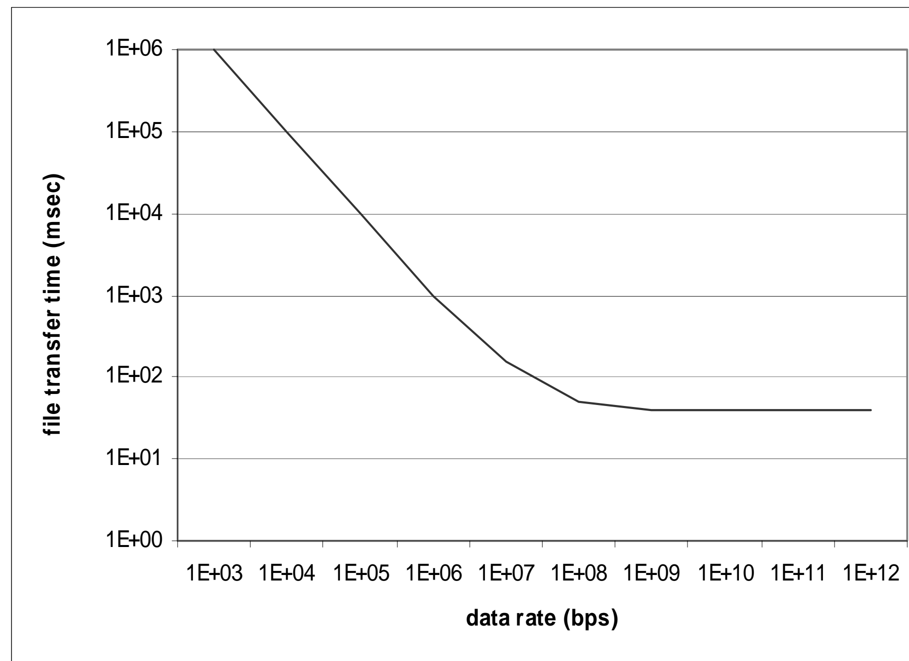

Figure 4: Network protocol overhead

2.1.6 Window of Opportunity

One of the assumptions for the techniques of progressive mobility described in this thesis is that the time to send a message (an application in our case) is directly proportional to the number of bytes sent and the data rate offered by the underlying network technology.

As we learn from Figure 3 and Figure 4 this is only the case for messages larger than 2 kByte and data rates almost up to 10 Mbps with a maximum of 1Gbps.

Aside from these limitations there is however a large window of opportunity where the preconditions of progressive mobility hold. The size of the applications on which we will apply the technique of anticipative mobility using pre-fetching of permuted code range in size from 65 kByte to 184 kByte, far above the minimum size of 2 kByte. The data rates in current wireless environments have an order of magnitude of 10 Mbps or lower. In 3G networks, the ambient intelligence environment that we have in mind for these progressive mobility techniques, a mobile may be granted 144 kbps when it is close to the base station with small shadow fading. But if the user is in the fade zone or fringe of cell, the data rate will drop considerably [Shaw-Kung Jong 2000].

2.2 Application

In this section we describe the relevant entities of the application dimension of the conceptual framework.

2.2.1 Internal Structure

Many implementations of programming languages, especially those that employ garbage collection, store their code and computational state as one chunk of memory. In this chunk we find all the elements necessary for the evaluation of the program. Again for many language implementations this is: a stack of some sort and a dictionary with the variable bindings and a representation of the abstract grammar. In chapter 5 we will take advantage of these properties to encapsulate the computational state of an application. Figure 5 shows the typical entities of a running application.

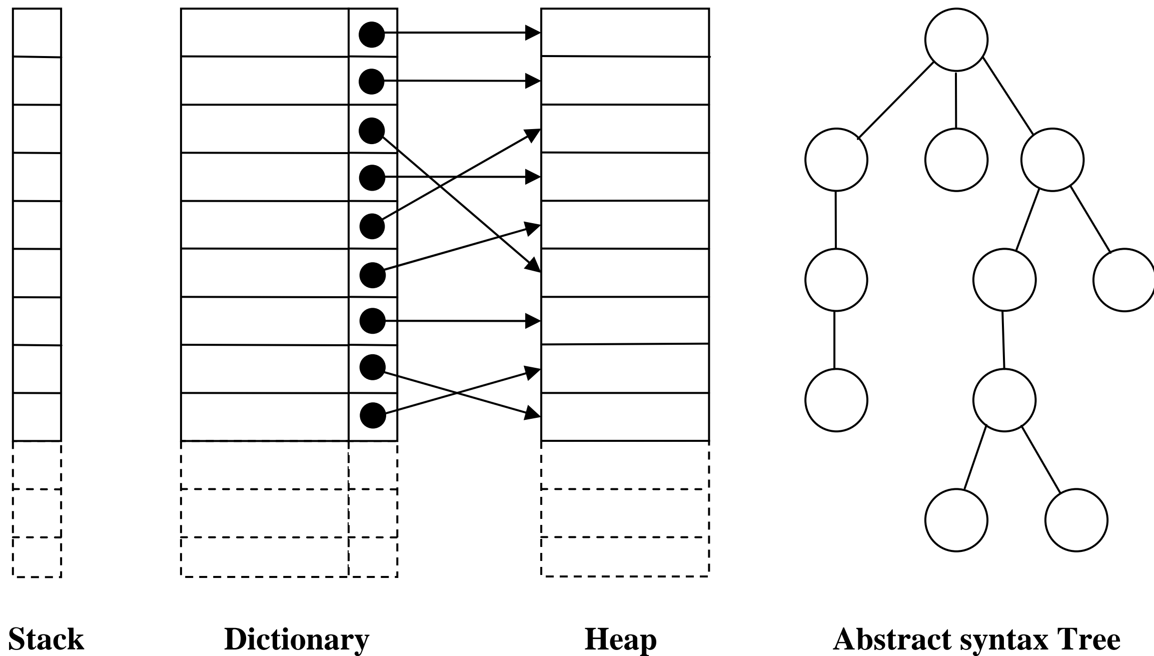

Figure 5: Typical entities of a running application

2.2.2 Size

The size of a mobile application may vary from a few bytes to several Mbytes. A typical environment for instance that is a likely candidate to gain from progressive mobility is the Multimedia Home Platform. This platform is currently investigated by our research group in a joint collaboration between Vlaamse Radio en Televisie (VRT, public broadcaster of Flanders), and IMEC (Interuniversity MicroElectronics Center) in one of the e-VRT projects funded by the Flemish government.

The Multimedia Home Platform accepts data (teletext, pictures, ...) and code (Java) over a shared 2Mbps channel called a carousel that provides on a regular basis, time slots for all the data and code to be transported to the set-top box on the television set. Typical MHP applications range from 60–300 kB. Browser applications range between 200–400 kB. An HTML compliant browser is not on the market yet but its expected size range between 700-800 kB, a number of bytes that is already too large for current set-top boxes which means that partitioning of the code will be necessary, not even to get an improved performance but just to get it up and running anyhow.

These are sizes and data rates that perfectly fit in our window of opportunity.

2.2.3 Granularity

Granularity is the relative size, scale, level of detail that characterizes an object or activity. In the context of this thesis, we will use the term granularity as an indication of the number of lines of code per usable unit. We will say, for instance, that the Java environment offers a granularity at the level of a class. It makes no sense to migrate a single Java method since the security mechanism of Java imposes that only complete classes can be loaded and started. Other programming environments as Smalltalk and Borg offer a much smaller granularity at the level of methods (Smalltalk) or even expressions (Borg).

2.2.4 Evaluation Time

Although the theoretical maximum network speed is enormous (see section 2.1.3), in practice we find that compared with the raw "number crunching" power of microprocessors where processor speeds of Gbps are common, the expected **3G** speed ranging from 144 kbps to 384 kbps [Tanenbaum 2003] is still several orders of magnitude slower if we compare their **bit processing speeds**. According to Moore's Law[6] [Moore 1965] data density on a chip doubles every 18 months and therefore also the chip's speed increases since the distance between the transistors is reduced. This is why CPU speeds are known to double almost every couple of years. We expect that this will remain the case through the end of this decade.

In some maritime environments data transmission rates of 2400 bps are still in use. And in the new standard Multimedia Home Platform (MHP), a digital video broadcasting standard intended to combine digital television with the Internet, we find a maximum data rate of 2 Mbps (256 kB/sec) for a channel that needs to share teletext, shop images, interactive TV data and java programs. In practice this means that if want to send an application via this channel we might expect a maximum data rate of 5kB/sec.

2.2.5 Delay

Delay in a computer system (**system latency**) is often used to denote any delay or waiting that increases real or perceived response time beyond the response time desired. Specific contributors to computer latency include mismatches in data speed between the microprocessor and input/output devices and inadequate data buffers. The possible time “wasted” by a (Just In Time) compilation step, evaluation delay caused by the operating system scheduling, concurrency problems etc.

In a mobile environment performance is measured by invocation latency. **Invocation latency** is the time from application invocation to when the evaluation of the program actually begins [Krintz et al. 1998]. It is the combination of the network latency and the system latency that occurs before the evaluation of the application starts.

The delay between the invocation of an application and the appearance of the user interface will be called: **User interface latency** in this dissertation. It is the sum of the network latency and the system latency that occurs before the appearance of the user interface.

Within a computer, latency can be removed or "hidden" by such techniques as pre-fetching (anticipating the need for data input requests) and multithreading, or using parallelism across multiple execution threads.

2.2.6 User Interfaces

In information technology, the user interface (UI) is everything designed into an information device which a human being may interact with, including display screen, keyboard, mouse, light pen, the appearance of a desktop, illuminated characters, help messages, and how an application program or a Web site invites interaction and responds to it. In early computers, there was very limited user interface except for a few buttons at an operator's console. The user interface was largely in the form of punched card input and report output.

---

[6] The observation made in 1965 by Gordon Moore, co-founder of Intel, that the number of transistors per square inch on integrated circuits had doubled every year since the integrated circuit was invented. Moore predicted that this trend would continue for the foreseeable future. In subsequent years, the pace slowed down a bit, but data density has doubled approximately every 18 months, and this is the current definition of Moore's Law, which Moore himself has blessed. Most experts, including Moore himself, expect Moore's Law to hold for at least another two decades.

Later, a user was provided with the ability to interact with a computer online and the user interface was a nearly blank display screen with a command line, a keyboard, and a set of commands and computer responses that were exchanged. This command line interface led to one in which menus predominated. And, finally, the graphical user interface (GUI) arrived, originating mainly in Xerox's Palo Alto Research Center, adopted and enhanced by Apple Computer, and deployed by Microsoft in its Windows operating systems and by UNIX adepts in X-Windows.

Applications that communicate with the user by a graphical user interface (GUI) spend a lot of time building this GUI (see also section 3.7) which may lead to large user interface latencies.

2.2.7 Predictability

Basic properties of program predictability – for both values and control – are defined and studied in [Sazeides 1998]. Program predictability originates at certain points during a program's execution, flows through subsequent instructions, and then ends at other points in the program.

For many applications, and especially those build with imperative programming languages, if we launch the application over and over again, its program flow after the start will always be the same for a certain amount of time. We call this time the predictable deterministic time zone. The process of building the graphical user interface is typically the same each time the application is started and thus largely predictable. As soon as the user interacts for the first time with the application, the program flow becomes less predictable.

To a lesser extent, many applications without a user interface also seem to follow a highly predictable process during startup until their first interaction with an unpredictable environment such as the connection with external systems, generation of a random number based on a real time seed delivered by the system clock etc...

As a final observation, typical source code contains a lot of low priority chunks for which it is predictable that loading can be deferred until the last moment. Class file splitting [Krintz 2001], partitions a Java class file into separate hot and cold class files where the low priority code is grouped in the cold class to avoid transferring code that is never or rarely used.

Examples of low priority code are:

- Exception handling (unless exeptions are used to structure the program flow)
- Code from abstract methods (e.g. in Smalltalk: `self subclassResponsibility`)
- Code from cancellation methods (e.g. in Smalltalk: `self shouldNotImplement`)
- Program parts that are not used in the predictable time zone
- Code that is only needed for testing and debugging purposes

2.2.8 Components

Composition is the act of applying a composition operator (that forms part of a composition model and theory) in a given context. Components are the subjects of composition. Composites (also called assemblies) are the results of composition. [Szyperski 2003].

A software component has to be a unit of deployment. Furthermore, to enable dynamic scenarios, it has to also be a unit of versioning and replacement. [Szyperski 2003]

Usually, a component provides a particular function or groups related functions. In programming design, a system is divided into components.

In object-oriented programming and distributed object technology, a component is a reusable program building block that can be combined with other components to form an application. Components can be deployed on different servers in a network and communicate with each other for needed services. A component runs often within a context called a container.

Coupling and cohesion are attributes that summarize the degree of interdependence or connectivity among subsystems and within subsystems. We can define cohesion in terms of intramodule coupling, normalized to between zero and one [Edward 2001].

An object-oriented information system is decomposed into entities; each entity is decomposed into classes of objects. Good object-oriented system design should exhibit **high cohesion inside entities and low coupling among entities**. In the research literature metrics are proposed for cohesion and coupling and can be used to define a quality metric at the system level [Nejmeddine 2002].

Component-based software development stands for software construction by assembling independent building blocks. Typical component models are the Component Object Model (COM) and JavaBeans. These models prescribe standards for the collaboration of independent applications, which should yield improved development productivity and, in particular, more adaptable software. Notable success has been reported for systems implementation in well-understood application domains, such as graphical user interfaces [Nierstraz 1995]. The building of the user interfaces in an object oriented environment is mostly delegated to one instance of a designated class.

The technique of progressive mobility splits existing applications into components that also should exhibit high cohesion inside entities and low coupling among entities. The components will constitute the parts that will be transported, progressive in time, from a sender to a receiver.

2.2.9 Distributed Systems and Applications

There is some mix-up in the literature between the notion of a computer network and a **distributed system**. A distributed system is a collection of independent computers that appears to its users as a single coherent system. Usually, it has a single model or paradigm that it presents to the users. Often a layer of software on top of the operating system, called middleware is responsible for implementing this model [Tanenbaum 2003]. A well-known example of a distributed system is the World Wide Web, in which everything looks like a document.

In this dissertation we describe **distributed applications.** Sometimes these applications become only temporarily distributed during the process of migrating component by component from a sender to a receiver.

2.2.10 Choosing an Experimental Programming Environment

The experiments conducted for this dissertation where performed in mainly three different programming environments. Our first choice was always **Smalltalk** for it allows fast prototyping and provides reflection, promising easy access to runtime structures e.g. the execution stack of processes. Smalltalk is a programming language, where the objects that define the language are themselves built with the language. Hence in Smalltalk, code entities such as classes and methods are themselves programmable and extensible objects, just like any other Smalltalk object. Smalltalk represents processes as objects. The VisualWorks Smalltalk environment that we employed comes with an add-on: **Opentalk** that provides an environment for the development and deployment of distributed applications.

As it turned out, Smalltalk did not always deliver as promised. Freezing a process object, migrate it to another host and restart the process over there was not as simple as we thought it should be. Sometimes, for small applications, we could make a workaround using **Boss** as serializer but for the rare occasions that Smalltalk let us down we moved to a language that was built for the migration of processes in the first place: **Borg**.

Finally, when the proof of concept was achieved we implemented as much as possible a similar system in a more widely adopted language: **Java**. In order to obtain sufficient support for our experiments we extended the basic Java environment with the **RMI** system and the **µCode** toolkit [Picco 1998].

2.2.11 Programming Languages Used

We describe these programming environments, their properties and frameworks used in alphabetic order:

2.2.11.1 Borg

Borg [Van Belle et al. 2001] is a mobile agent environment developed at the Programming Technology Lab of the Vrije Universiteit Brussel. Borg is an extension of Pico [D'Hondt 2003], a functional, dynamically typed, statically scoped language.

An *agent* is an active autonomous software component that is able to communicate with other agents. The term *mobile* indicates that an agent can migrate to other agent systems, while carrying its program code and data.

The Borg system provides a platform with active autonomous agents, which communicate with each other over a network, and which are able to migrate over this network. The Borg agents can be considered as mobile components. In general, components are not active entities but in the Borg context, a component is an active piece of code, which can communicate with other components on the network. A Borg component is able to migrate to other machines. A component's state can only be modified by sending a message to that component; all data of a component is private. Note that *component* is not the same as *object*. Components are active entities with independent private data. A component usually consists of a number of objects. We will interchangeably use the terms *component or agent* during the remainder of this paper. The term component indicates the fact that the entity is a part of a greater entity: the application. The term agent refers to the autonomous role of the entity. In the Borg environment, an application consists of a number of cooperating components.

Besides other, in this context less relevant properties, the Borg mobile architecture features:

- **Strong mobility**

A component can migrate between agent systems during its evaluation. Strong mobility is seldom found in current, Java-based, agent systems due to some technical drawbacks of Java. Because Borg has the ability to reify the complete computational state of a running process, including its runtime stack, strong mobility is one of its standard features.

- **An easy to use agent communication layer.**

The communication layer consists of a serializer and an objectcall-like syntax; it allows agents to pass messages to each other. Agents always communicate in an *asynchronous* fashion. The reasoning behind this design decision is the notion of being *autonomous*: an agent should be designed as a separate entity, sending messages to, and receiving messages from other agents, not as an entity which transfers its control flow to other agents.

- **A hierarchical naming / routing system**

Every agent has a human-readable name, which is always used to reference it. The naming system favors late binding, in the sense that we bind agents to each other at evaluation time, not at compile time, as we partially do with objects. There is no distinction between the name of an agent and the address of an agent. Instead of resolving the name of an agent to its place, messages are immediately routed to an agent based upon the receiver's name. This means, of course, that we need to change the existing communication infrastructure substantially. We no longer have a statically interconnected routing infrastructure and a separate, statically interconnected naming infrastructure; instead we have one hierarchical infrastructure in which we name agents and route messages between them.

- **A location-transparent distribution layer**

An agent can send messages to other agents, without having to know where the other agent resides. For example, if agent 'Alice' talks to agent 'Bob', and 'Bob' migrates to the agent system at the end of the universe, Borg keeps on routing messages between Bob and Alice using the shortest path between them. To provide this functionality the *name server and router are merged* into one entity.

- **Resource Transparency**

All resources in the mobile agent system (disks, user interfaces and so on) are represented as static agents (which cannot migrate). So whenever we migrate an agent, it stays connected to the resources it was using.

- **Garbage Collection**

A state of the art, highly performant garbage collector is incorporated into the system.

- **Synchronizing agents**

This is performed by using a rendez-vous between multiple agents. This rendez-vous can be in time and/or in space (synchronize at a certain computer). The primitives themselves are based upon CSP [Hoare 1985], with the exception that as guards, unification is used instead of sequenced statements.

2.2.11.2 Java

Java is a language that has been widely adopted for writing mobile code. The reason for this is the built-in support for a lot of features needed in mobile code. Java supports class serialization; this enables objects to be written to a serialized stream or to be read from a serialized stream into an object. Remote Method Invocation (RMI) [Sun 2002] is also a standard feature of Java; it allows a program to invoke methods of objects that exist on other Java Virtual Machines. It is possible to run a Java program on any machine that implements a Java Virtual Machine (JVM), because Java compiles its source-code into byte-code. This allows mobile code to be used on heterogeneous networks. Java also provides a class loader, a mechanism to retrieve and dynamically link classes in a running JVM, even from a remote location. This automatically supports weak mobility (see section 2.3.1.6).

RMI

The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM possibly on different hosts. RMI provides for remote communication between programs written in the Java programming language. RMI uses object serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

µCode

Even with all the advantages of Java, it has one major drawback: it is impossible to serialize the runtime stack which makes it impossible to make use of strong mobility in a standard Java environment. To overcome this limitation we deployed a mobile-code toolkit called µCode [Picco 1998], an extension of the Java environment. The basic operations provided by µCode enable creation and copy of thread objects on a remote µServer, and class relocation among µServers. A µServer is an abstraction of the run-time support and represents a computational environment for mobile threads. µCode supports synchronous and asynchronous invocation, as well as deferred and immediate evaluation of mobile code.

The unit of migration is the group, a container for classes and objects. Classes can be added to the group either individually or collectively by computing the transitive closure of a given class.

µCode contains a small set of abstractions and mechanisms that can be used directly by the programmer or composed in higher level abstractions for the creation of mobile code. It is written in Java to make it portable on all platforms. µCode is not a mobile agent system, but it focuses on mobility of code and state (Java classes and objects). µCode features a *CopyThread*-method, which allows us to copy a running thread, while keeping its internal state. Threads are objects in Java that allow concurrent programming; they have their own namespace, and appear to run as if they have the processor for themselves. By using the *CopyThread*-method, threads can be moved from one host to another.

µServers are µCode programs that form a layer between the Java code and the µCode programs. If we run µServers on two hosts, the available abstractions in µCode allow us to migrate a running thread from the sending to the receiving host. At arrival, the thread is restarted with the instance variables in the same state they had at the evaluation point where it was suspended before sending. This feature of µCode allows us to apply progressive mobility using component streams in a Java environment. There is no need to change the Java Virtual Machine or any standard libraries, so we preserve the full portability to all Java platforms.

2.2.11.3 Smalltalk

Smalltalk is a language, a complete class library, and an **interactive programming environment** rolled into one seamless whole. [Cincom 2003]. Smalltalk was developed at the Xerox Palo Alto Research Center in the '70s.

Smalltalk runs on a **virtual machine**, an abstract computer that can be implemented on different processors to provide a binary-portable execution environment. More recently, Java has popularized this implementation scheme.

In their desire to provide an interactive graphical environment the PARC Smalltalk group invented overlapping windows and pop-up menus, within the Smalltalk environment. Steve Jobs saw the Smalltalk environment at PARC in the eighties, took the ideas back to Apple and incorporated them in the Lisa and the Mac. The windowing environment pioneered in Smalltalk is now the common UI environment on most desktop computers.

Smalltalk performs automatic memory management (known as garbage collection), to relieve the programmer of the error-prone task of reclaiming unused storage.

Smalltalk introduced the notion of a **reflective programming language**, whereby the objects that define the language are themselves built with the language. Hence in Smalltalk, code entities such as classes and methods are themselves programmable and extensible objects, just

like any other Smalltalk object. Smalltalk even represents processes and method activations as objects. The use of Smalltalk objects to define the Smalltalk system itself allows the programmer to extend the language and environment.

Smalltalk is a dynamic implicitly-typed language where objects, not variables, carry type information, freeing the programmer from declaring variable types, but still providing complete type-safety [Cincom 2003].

Boss

The VisualWorks "Binary Object Streaming Service" (BOSS) is an important tool for converting most types of Smalltalk objects into a compact binary representation that requires relatively little memory space. Although BOSS is used mainly to store objects in and retrieve from a file, it can also be used for other purposes such as sending objects across a network. BOSS is implemented by a group of Smalltalk classes in category System-Binary Storage. BOSS writes objects onto a Stream: the Stream class provides a framework for a number of data structures, including input and output functionality, queues, and endless sources of dynamically- generated data. A Smalltalk Stream is similar to UNIX streams and provides a sequential view on an underlying resource; when reading or writing elements, the stream position advances until the end of the underlying medium has been reached.

Opentalk

Opentalk is a VisualWorks add-on that provides an environment for the development and deployment of distributed applications. Opentalk contains frameworks and components for creating or extending communication protocols, object services, remotely targeted user interfaces, remote development tools, and other architectural components common to distributed systems.

This Communication Layer consists of those components that define the base communication framework, several Smalltalk-to-Smalltalk communication protocols, and a select set of base services. It provides a set of frameworks and components for use by protocol developers who are creating protocol layers in VisualWorks, operating on top of either the TCP/IP or UDP transport layers.

Apart from being a distributed component "construction kit," the Opentalk Communication Layer provides a number of immediately useful components. There is a complete object request broker implementation that supports configurations using several kinds of object adaptors that exploit either TCP or UDP sockets. Brokers can be configured to use standard unicast communication or multicast and broadcast messaging. A set of basic services is also provided. Any application wishing to send or receive remote requests needs to create and maintain a request broker. Request brokers provide transparent remote communication between Smalltalk images and represent the communication layer to communicating applications.

The latest version of Opentalk (VisualWorks 7.1) provides four passing modes: **Pass-by-reference**, the default passing mode, **pass-by-value** for small objects, **pass-by-name** for entities with a more global scope such as Classes and **pass-by-OID** (Object IDentifier), a species of 'pass-by-name' for domain class instances. The standard passing mode can always explicitly overridden by another passing mode.

In the case of anticipative mobility and proactive migration and evaluation some selected objects are replicated to all involved potential receivers. In such cases, if a replicated object is an argument to a remotely invoked operation, it is a waste of resources to pass the replicate by either reference or value. Pass-by-reference entails remote message sends; pass-by-value entails the marshaling of a complete copy. In contrast, **pass-by-OID** allows anticipative migrated objects to be passed by no more than the object identifier under which they were pre-registered in the object tables of both the sending and the receiving object adaptors. A passed-by-OID object, on receipt, resolves to either (a) its local replicate or (b) an exception if the passed OID has not been pre-registered at both sending and receiving locales.

2.3 Techniques

In this section we describe the relevant entities of the techniques dimension of the conceptual framework.

2.3.1 Mobility

Mobility (the quality of moving freely) and migration (moving from one place to another) are natural processes and stem from a desire to move either toward resources or away from threats.

If we look at the entities available in **a computational process**, we distinguish the *host* environment, the *evaluator*, the *data*, the *code* and the *state* of the process itself. All of these entities, separate or grouped, are possibly subject to migration (Figure 6).

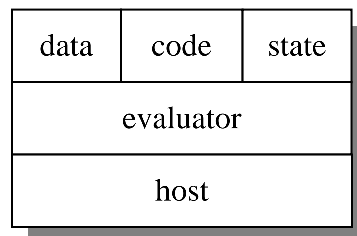

Figure 6: Entities subject to migration

2.3.1.1 Host Mobility

Portable computers, PDA's and wearable computers are intended to move by their own nature and by moving the host we also move its code, data, evaluator and process. Since they are contained in the host machine they are an integral part of it. Moreover, motion is relative, if we walk around with a PDA in our pocket all the host computers of the world are in a relative motion too. Some of the applications on these host machines need to be aware of their relative motion.

One of the goals of ubiquitous, pervasive computing is to embed more computers into our daily environment, and yet make them less noticeable. To do so, applications in these *smart spaces* or *intelligent environments* use information about the environment to adapt to the changing context in which they run. These applications need to be aware of their environment; they need to know the location of people and of their mobile devices, the current weather or traffic conditions, the status of computational and human services, and so forth.

Pervasive-computing applications must be aware of the context in which they run and *move*. These *context-aware* applications should be able to learn and dynamically adjust their

behaviors to the current context, that is, the current state of the user, the current computational environment, and the current physical environment, so that the user can focus on his current activity [Chen 2002].

2.3.1.2 Evaluator Mobility

A mobile evaluator seems a strange idea at first glance but they are common practice. If Java byte codes arrives in a Web browser they trigger automatically the launch of an evaluator for the byte codes, here a Java Virtual Machine. The evaluator code, available on the hard disk, is loaded to the memory to be evaluated by the underlying host machine. Sometimes if the evaluator code is not yet available on the hard disk it can be migrated over the internet to the host machine.

Some mobile code security mechanisms attach the evaluator to the code so as to be sure that the code will be evaluated with the correct non-tampered-with evaluator.

2.3.1.3 Data Mobility

Historically data was the first entity that became mobile. A large amount of data was stored on external memory as magnetic or optical cards, tapes or disks. This external memory was monitored by a powerful machine. The combination of the two is usually called a **Data**base. If a user needs some information he[7] would launch a query and the resulting data was migrated to the user. It leads to the very popular client-server paradigm.

Client-Server Paradigm: In this paradigm, see Figure 7, a server provides a set of services including access to resources like databases. The implemented code of these services is realized on the server and processed by the server. The server has the know-how, the resources, and the processor. The client uses the services provided by the server [Lange et al. 1998].

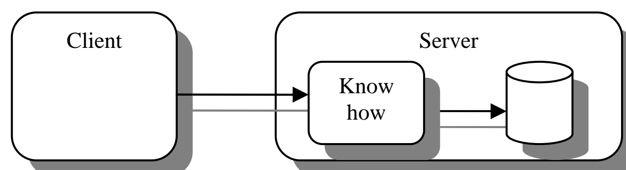

Figure 7: Client-Server Paradigm

2.3.1.4 Code Mobility

Sometimes code is hardwired, in the host machine. Often this is the case in embedded systems, a combination of computer hardware and software that is specifically designed for a particular kind of application. But most of the times before the code is evaluated it is loaded from a storage medium as a hard disk to a fast memory medium as DRAM to become evaluated. Although this kind of internal code transport is not called mobile code or code migration, several techniques described in this dissertation are also applicable to code loading.

In recent years we have witnessed the appearance of new paradigms for designing distributed applications where the application components can be relocated dynamically across the hosts in a network. This form of code mobility lays the foundation for a new generation of technologies, architectures, models, and applications in which the location at which the code is evaluated comes under the control of the designer, rather than simply being a configuration

[7] *he* should be read as *he or she* throughout this dissertation.

accident [Picco 2001]. It allows for new functionalities such as local resource access and dynamic load balancing.

Mobile code has an inherently dynamic character: Component interaction models need not be fixed at design time, but can change according to the states of the software environment. Mobility holds the promise of truly dynamical architectures, capable of changing their topological layout and their interactional properties in response to split-second changes in system requirements.

Security can be implemented less statically: mobile agents can go to secure locations when security is important, or can choose to roam in less strict **environments** when it is not - and a better performance can be gained. The concept of security in a system will move from being a static property to being a service that can be offered for those components that might need it at some point in time, and that will be ignored if the service is not needed.

Besides the fact that a mobile agent can move to resources to interact locally with them, the agent itself can also be considered a **computational resource**. As such, it might be summoned when it is needed in a local computation somewhere else in the system, and be dismissed afterwards.

Mobile code comes in many forms and shapes. [Fuggetta et al. 1998] Mobile code can be represented by *machine code*, allowing maximum evaluation speed on the target machine but in doing so sacrificing platform independence. Alternatively, the code can be represented as *bytecodes*, which are interpreted by a virtual machine (as is the case for Java in the Jini framework [Arnold et al. 1999] and Smalltalk). The use of intermediate bytecodes provides platform independence, a vital property in worldwide heterogeneous networks. The third option, which also provides platform independence, consists of transmitting *source code* or program *parse trees* [Franz and Kistler 1997].

Using source code as a transport medium in low data rate networks appears to be not such a bad idea at all: The same program representation contains less bytes if it represented by its source code instead of its compiled version since (1) It does not contain the libraries and other stuff added by the linker, (2) since a higher level language is more expressive than machine code; it allows: “saying more with less words”. (3) The source code in ASCII allows a high compression rate using standard compression algorithms.

To obtain maximum execution speed an extra compilation step is necessary after the transmission but the speed of the compilation process that generates native machine code is typically much faster than the transportation time of the resulting byte codes over a low-data rate network .

In what follows we describe three mobile code paradigms.

Code-on-Demand Paradigm

According to this paradigm, see Figure 8, you first get the know-how when you need it. A host, called the client, gets the code from another host, called the server, when it needs it. The client holds the processor capabilities and the local resources, but does not need preinstalled code. The server has the know-how and resources. Java Applets are typical examples. Applets get downloaded in Web browsers and evaluate locally [Lange et al. 1998].

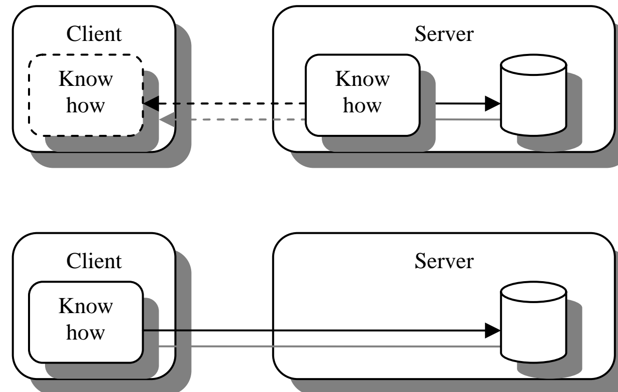

Figure 8: Code-on-Demand Paradigm

Remote Invocation Paradigm

According to this paradigm, see Figure 9, the know-how is sent to a server for evaluation. A host, called the client, sends the code to another host, called the server, when needed. The server holds the processor capabilities and the local resources, but does not need preinstalled code. The client has the know-how. Java Servlets are examples if they get uploaded to remote servers and evaluate there [Carzaniga et al. 1997].

This paradigm can also be viewed as a special case of the code on demand paradigm [Lange et al. 1998] where it is the server that "asks" for the code.

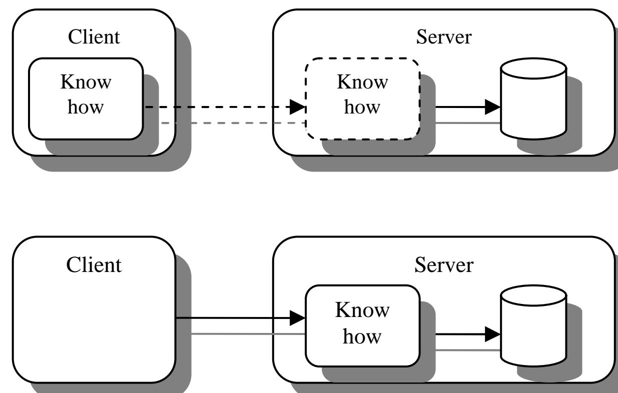

Figure 9: Remote Invocation Paradigm

Mobile Agent Paradigm

Clients and server merge to hosts. Any host in a network holds any mixture of know-how, resources, and processors. The code in form of mobile agents is not tied to a single host, but is available throughout the network [Lange et al. 1998]. See Figure 10.

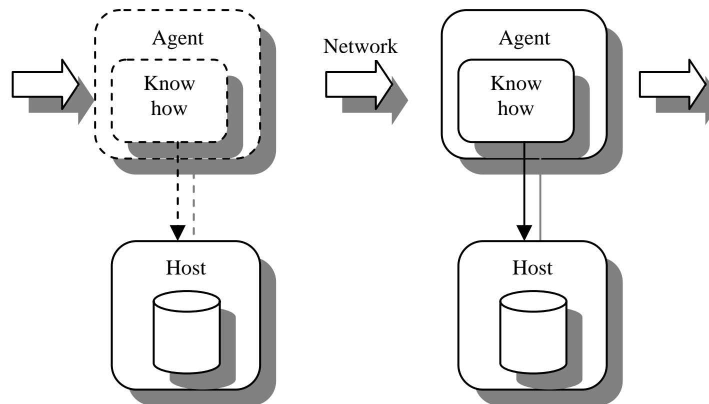

Figure 10: Mobile Agent Paradigm

Mobile code and mobile agents hold the potential to share the next generation of technologies and models for distributed computation. The first steps of this process are already evident today: Web applets provide a case for the least sophisticated form of mobile code, Java-based distributed middleware makes increasing use of mobile code, and commercial applications using mobile agents are operational [Picco 2001].

The term *agent* refers to the function of the mobile code i.e. the task of representing a query or a service for a human or non-human client. Also the term *mobile objects* is often used to indicate that code, data, and a thread is moved. In this thesis we will use the term agent only if we want to focus of its representative function, but mostly we make abstraction of the current task and structure and use the more generic term **mobile code.**

2.3.1.5 Process Mobility

A process is an instance of a program running in a computer. It is started when a program is initiated and is characterized by a state and an address space trough which the code and data can be accessed. At the machine level the state contains at least the program counter, the program status (flags) a stack pointer and the general registers.

Mobile agents that move as a process are called strong migration agents in contrast to weak migration agents where only the code is moved from one host to another and restarted from scratch. [Fuggetta et al. 1998].

Not all high level programming environments provide the necessary reflection to grant the application access to its own computational state, but if this level of reflection is available we may apply similar approaches on process mobility as the one we find at lower levels in the computer architecture hierarchy e.g. the operating systems.

A process is a key concept in **operating systems**. It consists of data, variable storage, and the state specific to the underlying Operating System (OS), such as parameters related to process, memory, and file management. A process can have one or more threads of control; threads, also called lightweight processes, containing their own variable storage, but share a process's address space and some of the operating-system-specific states, such as signals.

Migration performance of operating systems depends on initial and run-time costs introduced by the act of migration [Milojicic et al. 1999]. The initial costs stem from state transfer. Instead of at migration time, some of the state may be transferred *lazily* (on-demand), thereby introducing extra run-time costs. Both types of cost may be significant, depending on the application characteristics, as well as on the ratio of state transferred eagerly/lazily.

If only part of the process state is transferred to another node, the process can start its e valuation sooner, and the initial migration costs are lower. This principle is called *lazy evaluation*: and was also applied in our pre-fetched code experiments. Actions are not taken before they are really needed with the hope that they will never be needed.

However, when everything is needed, penalties are paid for postponed access. For example, it is convenient to migrate a huge address space on demand instead of eagerly. In the lazy case, part of the space may never be transferred if it is not accessed. A process address space usually constitutes by far the largest unit of process state; not surprisingly, the performance of process migration largely depends on the performance of the address space transfer.

Various data transfer strategies have been proposed in order to avoid the high cost of address space transfer [Milojicic et al. 1999].

• The **eager (all)** strategy copies all of the address space at the migration time. Initial costs may be in the range of minutes.

• The **eager (dirty)** strategy can be deployed if there is remote paging support. This is a variant of the eager (all) strategy that transfers only modified (dirty) pages. Unmodified pages are paged in on request from a backing store. Eager (dirty) significantly reduces the initial transfer costs when a process has a large address space.

• The **Copy-On-Reference (COR)** strategy is a network version of demand paging: pages are transferred only upon reference. While dirty pages are brought from the source node, clean pages can be brought either from the source node or from the backing store. The COR strategy has the lowest initial costs, ranging from a few tens to a few hundred microseconds. However, it increases the run-time costs, and it also requires substantial changes to the underlying operating system and to the paging support.

• The **flushing** strategy consists of flushing dirty pages to disk and then accessing them on demand from disk instead of from memory on the source node as in copy-on-reference. The flushing strategy is like the eager (dirty) transfer strategy from the perspective of the source, and like copy-on-reference from the target's viewpoint. It leaves dependencies on the server, but not on the source node.

• The **precopy** strategy is used to augment the process availability in the same sense as proactive migration. It reduces the "freeze" time of the process, the time that process is neither evaluated on the source nor on the destination node. While the process is evaluated on the source node, the address space is being transferred to the remote node until the number of dirty pages is smaller than a fixed limit. Pages dirtied during precopy have to be copied a second time.

2.3.1.6 Weak and Strong Mobility

If an executable component is migrated before the application has started it suffices to send over its code and start it up in the same manner as applets are loaded to a web browser and started. If, however, the application was already running before migration, one should send not only the bare code but also the intermediate values of the local variables of that evaluation unit and the information of the exact point in evaluation where the entity was stopped to be able to resume it at the same point. This extra information is usually referred to as *the computational state and runtime stack*. This kind of migration is known as *strong mobility* while the former is called *weak mobility* [Fuggetta et al. 1998]. In the remainder of this dissertation we will refer to the computational state to indicate the values of all the variables of the application **including the runtime stack**.

A typical way of moving an application from host to host is composed of the nine sequential steps from Table 1 (page 26). Strong and weak mobility only differ in the way the current state of the process is packed and unpacked. In strong mobility the computational state and runtime stack is contained in the same package, in weak mobility the state (or part of it) is passed by parameters under control of the programmer.

2.3.2 Server - Push versus Client - Pull

The dominant paradigm of communication on the world-wide web and in most distributed systems is the request-reply or client-pull model. In this model of distributed information systems, a client actively "pulls" information from the server. Since the early days of the Internet, systems such as electronic mail and Usenet News have attempted to overcome the deficiencies of this pull model by allowing producers of information to "push" their information closer to the clients. In the push model, an information producer announces the availability of certain types of information, an interested consumer subscribes to this information. The producer periodically publishes the information (pushes it to the consumer).

Several reasons motivate the need for push systems. The most important one is that the WWW is based on a simple request/reply scheme [Berners-Lee 1996] that requires the user to issue a request whenever he needs information. This imposes a "synchronous" interaction scheme, whereas push systems allow asynchronous information distribution: Ideally, whenever information of the user's choice becomes available it gets distributed [Hauswirth 1999].

The idea behind code-on-demand was the thin client or network computer. In this view a workstation should only contain the bare minimum of the operating system, possibly in a non-volatile ROM, all the extra software that is needed is loaded from the network, in the same sense that Java Applets are loaded and evaluated on the client. The main advantage of this scheme is that it allows maintaining a centralized codebase and that installation and maintenance problems of the clients are virtually eliminated. Software update mechanisms become very simple since they only need to adapt the central code base. Larger applications may apply dynamic binding of lean software [Wirth 1995] where, as an example, the help dialog is only loaded if activated.

The main disadvantage of this setup is that interoperable code always needs to be available behind the scenes and that long delays may be expected at start up. Moreover a client-pull setup will always need an extra communication step to send the request to the server. This makes the pull strategy inherently slower than the push strategy we adopted for progressive anticipative mobility using pre-fetching of permuted code.

A combination of the two paradigms is possible however. If a pre-fetching scheme is applied, then, once the predictable time zone is surpassed and the not all the code is loaded yet, the server has to rely on statistical data to predict the next chunk of code that is needed at the client. This is the time that the client can come to help by giving some hints to the sender. If the client communicates to the sender the exact place in the code it is halted, at a semaphore or another synchronization mechanism, the sender might be able to determine the appropriate next chunk of code.

2.3.3 Parallelism

Actual computer architectures provide separate processors for input/output (code loading) and main program evaluation. Disk controllers, modems and network controllers contain their own processing units. This enables us to send code to another host in parallel with the evaluation of the main application. We also see new upcoming techniques that favor the use

of parallelism e.g. hyper-threading technology [Intel 2002] which enables thread-level parallelism by duplicating the architectural state on each processor, while sharing one set of processor evaluation resources. When scheduling threads, the operating system treats the two distinct architectural states as separate "logical" processors. This allows multi-processor capable software to run unmodified on twice as many logical processors. While hyper-threading technology will not provide the level of performance scaling achieved by adding a second processor, benchmark tests show that some server applications can experience 30 percent gain in performance [Intel 2002]. We now already witness the introduction of high performance architectures as the Cray MTA-2 processor. Each MTA processor has up to 128 RISC-like hardware threads. Each thread is a hardware stream with its own instruction counter, register set, stream status word and target and trap registers. In a few years we may expect up to 20 different threads running on one standard microprocessor chip.

To take full advantage of possible parallelism one is compelled to split up a problem in independent threads.

Amdahl's Law [Amdahl 1967] is a law governing the *speedup* of using parallel processors on a problem, versus using only one serial processor.

In chapter 4 we propose a technique that requires the division in components of an existing application. Since we have to split our application in different components anyhow we might apply existing techniques devised for parallel programming [Attali et. al 2000, Caromel et al. 1998] to split up the application in different components. Moreover, it might be advantageous to obtain concurrent processes at the same time. We will report on our experiments in section 4.6.5.2.

2.3.4 Reflection

Reflection is the ability for a program to manipulate its state and behavior as data during its evaluation [Maes 1987]. There are two aspects to reflection:

- **Introspection:**
  Makes it possible for an application to observe and reason about its state and behavior.
- **Intercession:**
  The ability for the application to modify its evaluation state or alter its interpretation or semantics.

We will rely on reflection to be able to capture the computational state of our applications, mostly to implement strong migration. This level of reflection is not offered by most classical programming environments, this why we will make use of languages as Borg, Smalltalk and special toolkits for Java.

2.3.5 Compression

File compression is used to hide network latency by decreasing the number of bytes transferred through the use of compact encoding mostly without the knowledge of the type of underlying data. The resulting size of compressed files (compression ratio) is dependent upon the complexity of the encoding algorithm. Typically, a similar complexity is required for decompression of the file prior to its use. Techniques with a high compression ratio necessarily need more time to decompress. Techniques with fast decompression rates are unable to achieve aggressive compression ratios.

Since compression techniques must trade of compression ratio for decompression time the latter must also be considered a source of invocation latency since it occurs online while the program is executing.

To minimize this latency, a compression technique should be selected based on the underlying resource performance, network, CPU, etc… Moreover since such performance is highly variable [Wolski 1998] selection of the best compression algorithm should be able to change dynamically. Such adaptive ability is important since the selection of a non-optimal format may result in substantial total latency.

In order to address this selection problem Krintz [Krintz 2001] introduces Dynamic Compression Format Selection (DCFS) a methodology for automatic and dynamic selection of competitive compression formats. Using DCFS, mobile programs are stored at the server in multiple compression formats. DCFS is used to predict the compression format that will result in the smallest amount of network latency given the data rate predicted to be available when transfer is triggered.

Code compression is another way to reduce overhead introduced by network delay. Several approaches to compression have been proposed to hide network delay in mobile code environments. J. Ernst [Ernst et al. 1997] describes an executable representation that is roughly the same size as gzipped x86 programs and can be interpreted without decompression. M. Franz [Franz and Kistler 1997] describes a compression format called Slim binaries, a compression scheme based on adaptive methods such as LZW [Ziv and Lempel 1977], but tailored towards encoding abstract syntax trees rather than character streams. It takes advantage of the limited scope of variables in programming languages, which allows to deterministically prune entries from the compression dictionary, and uses prediction heuristics to achieve a denser encoding.

The technique of code compression is orthogonal to the techniques proposed in this paper, and can be used to further optimize our results.

2.3.6 Reordering and Pre-fetching

One way to avoid invocation delay is to ensure that only those methods that will be executed are transferred across the network. Sirer et al. [Sirer 1999] describes such an optimization based on repartitioning of Java applications into modules that utilize network data rate more effectively. Other Java based techniques, Class File Splitting and Pre-fetching and Non-Strict Execution for Mobile Programs are proposed by Krintz [Krintz et al. 1999, Krintz et al. 1998].

2.3.6.1 Profiling

To determine the optimal ordering of code so that pre-fetching becomes possible, a more thorough analysis of the code is needed. This can be done either statically, using control flow analysis, or dynamically, using code instrumentation. Both techniques are empirically investigated in [Krintz et al. 1998] to predict the first use ordering of methods in a class. These techniques are directly applicable to our approach as well. More sophisticated techniques for determining the **most probable path** in the control flow of a program are explored in [Jason and Patterson 1995].

The static and dynamic profiling techniques used to determine the hot and cold parts can also be used to measure the dynamic behavior of our application. The obtained profile will allow us to predict ideal proactive migration points so that possible complications with sudden large memory allocations (section 5.6.4) can be avoided.

2.3.6.2 Class File Splitting and Pre-fetching

Reordering of code and data is also essential for reducing transfer delay. One possibility to accomplish this is by splitting Java code (at class level) into hot and cold parts [Krintz et al. 1999]. The cold parts correspond to code that is never or rarely used, and hence loading of this code can be avoided or at least postponed.

Class file splitting partitions a class file into separate hot and cold class files, to avoid transferring code that is never or rarely used. Class file splitting helps reduce the overall transfer delay and invocation latency. Invocation latency is the time required to begin evaluation of a program. In Java, this includes the time for transfer and load as well as any additional file processing required by the evaluation environment, e.g. verification.

Class file pre-fetching inserts pre-fetch commands into the bytecode instruction stream in order to overlap transfer with evaluation. The goal is to pre-fetch the class file far enough in advance to remove part or all of the transfer delay associated with loading the class file.

Java class file splitting was proposed by T. Chilimbi [Chilimbi et al. 1992] to improve memory performance. The goal of their research was to split infrequently used fields of a class into a separate class.

The splitting algorithm relies on profile information of field and method usage counts. With the profile information as input, a static bytecode tool performs the splitting.

Class file pre-fetching is an optimization that is complementary to class file splitting. Pre-fetching class files masks the transfer delay by overlapping transfer with computation, i.e., class files are transferred over the network while the program is evaluated. In the optimal case, this overlap can eliminate the transfer delay a user experiences. Effective pre-fetching requires (1) a policy for determining at what point during program evaluation each load request should be made so that overlap is maximized, and (2) a mechanism for triggering the class file load to perform the pre-fetch.

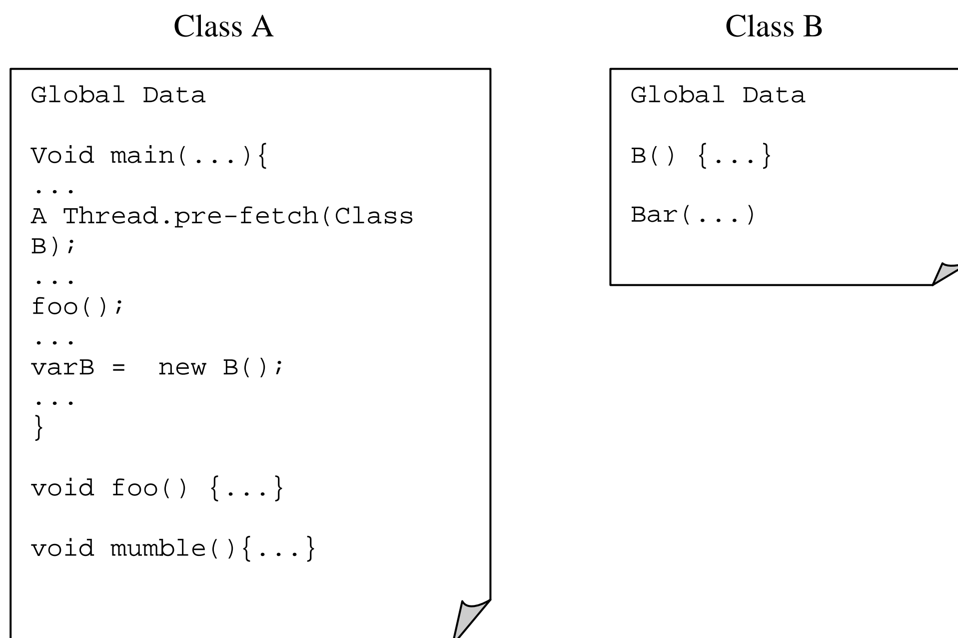

Figure 11: Splitted classes

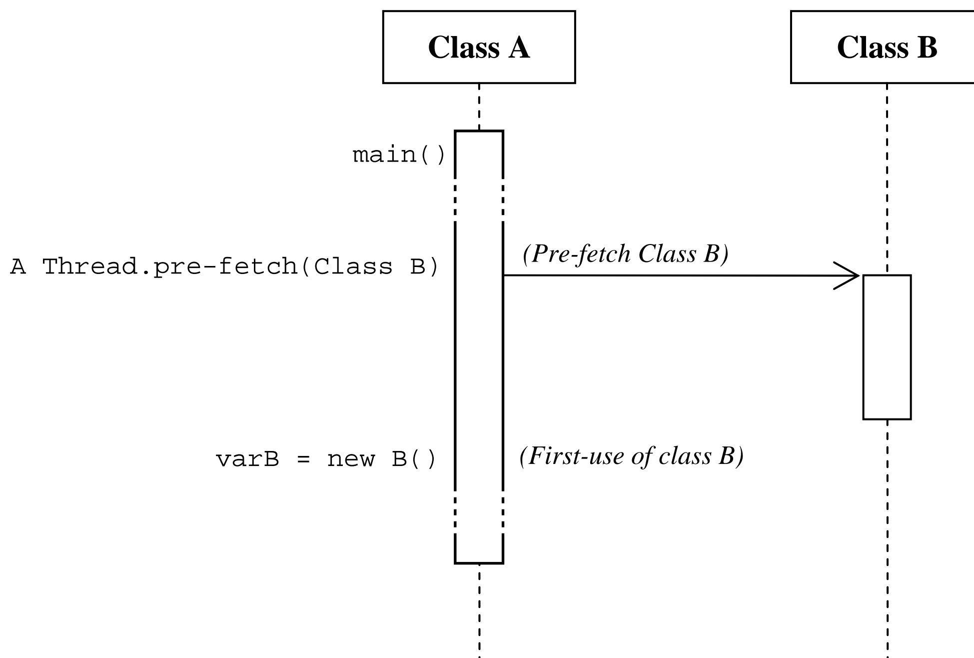

Figure 12: Class pre-fetching

Figure 11 and Figure 12 show the benefit of splitting and pre-fetching, The first class to be transferred is class A, and evaluation starts with the main routine. While executing main, a pre-fetch request initiates the loading of class B. We insert a pre-fetch request for class B, since it is needed when the first-use for class B is evaluated at the new B() instruction in main. If the evaluation of class A, after the pre-fetch and before this first reference to class B, takes more time than the transfer of B, the statement new B() will evaluate without waiting on the transfer of B. On the other hand, if there are not enough useful compute cycles to hide class B's transfer (that is, the time to transfer class B is greater than the number of cycles evaluated prior to A's instantiation of B), then the program must wait for the transfer of class B to complete before performing the evaluation of new B(). In either case, pre-fetching reduces the transfer delay since without pre-fetching evaluation stalls for the full amount of time necessary to transfer class B.

With verified transfer, class file splitting reduces the startup time by 10% on average. Without code verification, the startup time can even be reduced slightly more.

2.3.6.3 Non-Strict Execution for Mobile Programs

Overlapping execution with transfer using non-strict execution [Krintz et al. 1998] was proposed and simulated to parallelize the processes of loading and compilation/evaluation, a technique that is also adopted by this dissertation. To reorder the procedures in a first-use mode they used as a first approach static program estimation to predict the order of invocation for procedures, a second approach uses first-use profiling to create a profile indicating the order of invocation.

Static First Use Estimation uses a static call graph. To obtain the ordering, they construct a basic block control flow graph for each procedure with inter-procedural edges between the basic blocks at call and return sites. The predicted static invocation ordering is derived from a modified depth first search (DFS) of this control flow graph, using a few simple heuristics to guide the search. A flow graph is created to keep track of the number of loops and static

instructions for each path of the graph. When generating the first-use ordering, they give priority to paths with loops on them, predicting that the program will evaluate them first.

If real-time profiling is not possible or practical this approach can be used instead.

Profile Guided First Use Estimation uses profile information to determine the first-use ordering of procedures. A first use profile is generated by keeping track of the order in which procedures are invoked during a program's evaluation using a particular input.

All procedures that are not evaluated are given a first-use ordering during placement using the static approach described above. Since a program's evaluation path may be input dependent, they attempt to choose adequate sets of inputs in order to provide an evaluation path that is similar to most of the possible inputs.

Krintz et al. also proposes different transfer strategies that can take advantage of non-strict execution and program restructuring:

Parallel File Transfer can made optimal use of the available data rate. Current Internet HTTP transfer technology allows multiple files to be transferred in parallel. The HTTP 1.1 specification uses a single TCP connection that allows up to four transfers in parallel. To take advantage of this they model the transfer of multiple classes at once to assure that methods arrive as near to the predicted start of their evaluation as possible. The transferring files split the fixed amount of data rate available equally. Since data rate is shared, a schedule is required that indicates when class files should be transferred to obtain efficient overlap of the evaluation with transfer. A transfer schedule is created using the first-use procedure order determined by the reordering techniques. There are many factors that must be taken into account when developing a transfer schedule. First, information about the size of each procedure and class file is required. The size of global and local data is also needed. With the size information, the scheduler can make an informed prediction of the time it will take to transfer the various parts of each file.

Interleaved File Transfer groups Java class files to speed up the loading process [Krintz et al. 1998]. In Java, an application is composed of multiple classes each containing global data, local data and code. This organization is similar to other programming languages for which multiple files comprise the executable program: for those languages, the final program is typically a single binary. With interleaved file transfer, we consider a group of Java class files and compose a program as a single entity (an interleaved file), consisting of multiple procedures and data.

This technique transfers the procedures and data to the destination in the order specified in this virtual interleaved file. An interleaved file is a reordering of procedures. The transfer algorithm takes the application and the restructuring information as input. It generates an interleaved file from the input information and transfers it in the order dictated by the restructuring, e.g., methods from different classes may be interspersed for transfer. This transfer technique assumes that transfer proceeds at the method (procedure) level, in the order established by the restructuring algorithms.

2.3.7 Progressive techniques

The Interlaced Graphics Interchange Format (GIF) [Siegel 1996] is a format that tries to exploit the combination of low data rate channels and fast processors. An **interlaced GIF** file contains a picture that seems to arrive on your display like a fuzzy outline of an image that is gradually replaced by three successive waves of bit streams that fill in the missing lines until the image appears at its full resolution. Among the advantages for the viewers, using low data rate connections, are that the wait time for an image seems less and the viewer can sometimes

get enough information about the image to decide to click on it or move elsewhere. In the latter case the technique behaves as a form of data compression. Interlaced GIF is a widely used technique for speeding up image rendering on the Internet. Compression and rendering via the quadtree data structure called BCQ Progressive Image Transmission [Dürst 1997] seems to give even better results. Other progressive transmission methods are fractal images [Ghim and Chorng 2001] and **progressive JPEG**.

A **stream** is a data structure that is accessible as a contiguous sequence of data units representing a stream of data, transmitted continuously over a communications path. If the data units encode audio or video signals we call this audio and video streaming. Streaming media consists of a sequence of images; sound or both that are transmitted in compressed form and played on the receiving computer as they arrive. With streaming media, a user does not have to wait to download a large file before seeing the video or hearing the sound. The encoding of the media is often combined with compression (section 2.3.5). A frequently used algorithm for compressing video data follows the MPEG standard [Le Gall 1991].

The main characteristic of these transmission schemes is that the processing of the digital stream is started long before the load phase is completed. These techniques which send data progressive in time has inspired us to develop techniques that send mobile applications progressive in time so that its evaluation can be started long before the load phase is completed.

2.3.8 Other related techniques

The idea of progressive anticipative mobility using pre-fetching of permuted code was inspired by mobile agent research at our lab where agents are represented by **parse trees**. [Van Belle et al. 2001]. The migration of an agent happens by migrating the visited nodes in the parse tree during a transitive closure. We noted that if we could permute the order of the migration of the nodes, the evaluation of the receiving agent could start before the whole parse tree was sent over. This increased the virtual migration speed. In a JIT compilation environment a parse tree representation may have other advantages. Not only does it allow higher level, domain specific, more efficient compression techniques, but the tree preserves the control-flow structure of the original program, making it much easier to perform code optimizations.

A similar technique, although at a lower level of the computer architecture, where parallelism is exploited to speed up the processing of instructions is known as **pipelining**. With pipelining, the computer architecture allows the next instructions to be fetched while the processor is performing operations, holding them in a buffer close to the processor until each instruction operation can be performed. The staging of instruction fetching is continuous. The result is an increase in the number of instructions that can be performed during a given time period.

With progressive anticipative mobility using pre-fetching of permuted code the evaluation of the code is triggered by the loading process, since it is the loading process that switches the semaphores. There is a similar but reverse relation between evaluation and code loading in the **code on demand** paradigm. Java for instance provides a mechanism, the class loader, to retrieve and link dynamically classes in a running Java Virtual Machine. The class loader is invoked by the JVM run-time when the code currently in evaluation contains an unresolved class name. Although the goals of pre-fetching and code on demand are the same (they both aim to reduce transfer delay), both techniques are complementary since they can be employed at the same time. If in a pure code on demand setup there is not an immediate demand for new

code after the loading of a previous part, empty time slots will be the result. This leads to a less efficient loading process.

Another well-known example of code on demand is the use of **dynamic link libraries**. A dynamic link library (DLL) is a collection of small programs, any of which can be called when needed by a larger program that is running in the computer. The advantage of DLL files is that, because they don't get loaded into random access memory together with the main program, space is saved in RAM.

Continuous compilation [Plezbert and Cytron 1997] is a technique where interpretation, compilation to native-code and native-code evaluation are intertwined. The goal is to have the native-form available by the time the call to it occurs. The system essentially uses two threads of control: one thread compiles interpreted code into native-code form, while the other thread handles program evaluation of the interpreted and compiled code. The technique is complementary with progressive anticipative mobility using pre-fetching of permuted code.

Continuous compilation and **ahead-of-time compilation** are techniques that are typically used in a *code on demand* paradigm, such as dynamic class loading in Java. The goal of both compilation techniques, explored in [Krintz et al. 1999] and [Plezbert and Cytron 1997], is to compile the code before it is needed for evaluation. Again, these techniques are complementary to our approach, and can be exploited to further optimize our results.

Partial evaluation provides a unifying paradigm for a broad spectrum of work in program optimization, compiling, interpretation and the generation of automatic program generators [Jones 1996]. Although the name suggest similarities with progressive mobility where also only parts of the code are evaluated, partial evaluation is basically a program optimization technique and should perhaps better called: *program specialization*.

Much partial evaluation work to date has concerned automatic compiler generation from an interpretive definition of a programming language, but it also has applications to scientific computing, logic programming, metaprogramming, and expert systems.

Program slicing is a source-to-source transformation technique that is useful in construction, analysis, testing and debugging of programs [Venkatesh 1991]. A program slice contains the portion of a program that captures some subset of the program behavior. Algorithms are available for constructing slices for a particular evaluation of a program (dynamic slices) as well as to approximate a subset of the behavior over all possible evaluations of a program (static slices). The technique may be used for the refactoring of an existing application in mutually independent components.

3 Progressive Anticipative Mobility using Pre-fetching of Permuted Code

Trust no one. Always shuffle the cards yourself.
-- Unknown.

3.1 Abstract

In an ambient intelligence environment, featuring ubiquitous communication that enables everyday objects to communicate with each other and the user by means of ad-hoc and wireless networking there will be also the need to migrate code.

This migration will not be straightforward as in current stable networks since the timeframe in which we can send the code will be unpredictable. Therefore we need some kind of mechanism to break up code into smaller parts and send them one by one to increase the chance that it will fit in the current timeframe.

Precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately ready for evaluation at the receiving object.

We explore the possibility to harness the implicit parallelism found in the most simple network connections by starting the evaluation on the receiver in parallel with the migration of the last part of the code.

From the perception of the user the application is up and running much faster than expected. Network latency is hidden.

Streaming audio and streaming video are multimedia techniques that employ progressive transmission of encoded data and start the processing of the digital stream long before the load phase is completed.

In this chapter, we explore the idea of progressive transmission of software code instead of other media. The processing of the digital code stream can start before the load phase is completed by anticipating on the sequence of evaluation.

In this chapter, where we provide a proof of concept of the technique of pre-fetching permuted code, we start by presenting a technique to permute source code based on a profiling and reorder process. Then we can make use of the resulting data structure to implement a pre-fetching scenario in order to hide network latency and reduce system latency at the same time. We present the results of four experiments and discuss the results.

Roadmap:

- Introduction
- Basic Observations, assumptions and restrictions
- Profiling and reordering
- Reordering algorithm
- Pre-fetching
- Experiment to hide network latency
- Results
    - Benchmark
    - CoolImage
    - Gremlin
    - Adapted Gremlin
- Discussion
    - Speedup
    - Application Speedup versus Data rate
    - Pre-fetching Guidelines
    - Dealing with Semaphores
    - Applicability in other Environments
- Summary and Conclusion

3.2 Introduction

In an ambient intelligence environment, we will need to migrate code from one object to another in order to address some key issues as context dynamics or system adaptivity. The width of the timeframe available in such an environment to migrate the code depends on the movement of the components relative to each other and the reach of their wireless transceivers. The width of this timeframe will not be predictable, so we need some kind of mechanism to break up code into smaller parts and send them one by one, progressively in time, to the receiver. This will increase the chance that they will fit in the temporal time frame and that we will be able to migrate the complete block of code.

A disadvantage of this approach is that it may take quit some time before the complete block of code is transmitted, so precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately usable (ready for evaluation) at the receiver's end.

For the users perception it is also important that the user does not have to wait to long between its request for a service and the first perceptible reaction of the system. Therefore, we migrate first all the code that builds the user interface. We migrate this code in a format that allows the system to start the evaluation of the code even if it is only partial available.

In this first theme it is our goal to build a proof of concept of a system that breaks up code into smaller parts and sent them one by one, progressively in time, in a setting that supports

weak mobility. We will handle progressively strong mobility in chapter 4 and 5. The feasibility of the technique is validated by implementing prototype tools in Smalltalk.

Our contributions here are:

- Automatic permutation of compilable units based on a dependency graph so that the static structure of the representation of the application in a file structure reflects its dynamic behavior
- Introduction of a synchronization mechanism to support evaluation of code that is only partially present
- Pre-fetching of permuted code with focus on user interface latency

An important question is: what should be the ideal unit of code to be split into? We propose to use as unit those program abstractions that the code was built from. For example, in Smalltalk likely candidates at different levels of granularity would be: statements, methods, method categories, classes, class hierarchies, class categories, etc…

We like to take the unit of code as small as possible to achieve a high flexibility in the possible places where the code can be split, which in turn will make it possible to obtain a high level of parallelism. The smallest level in source code is a statement or the sending of a message. In many languages, statements or messages are aggregated in higher level structures (functions or methods), in such a way that if one statement of the function or one message sent in a method is being evaluated, the other statements or methods from that aggregate will often soon follow. Therefore taking the level of functions or methods as unit of code seems more appropriate. Especially for well-written object oriented programs, adhering to the good programming practice of keeping methods small, the splitting flexibility should remain high.

This is not possible in all languages. In Java for instance the smallest unit for code loading and therefore for code migrating is the class. In addition, the Java security model prescribes that this class file should contain al its methods and a security stamp to allow class file validation. In Smalltalk a much finer granularity is possible at the level of compilable units. We will split up our Smalltalk sources in methods, class descriptions, namespaces, window specs etc…

We describe progressive anticipative mobility using pre-fetching of permuted code, a technique that permutes the application at the granularity of the method level and exploits parallelism between loading and evaluation of code to hide network latency. It may allow many applications to start their evaluation early, especially programs with a predictable, deterministic startup phase (such as building a GUI). The technique allows us to start up the code before it is completely loaded. The underlying technique is also known as interlaced code loading [Stoops et al. 2002] and non-strict execution [Krintz et al. 1998] where the technique was simulated in a Java environment.

The feasibility of the technique has been validated by implementing a prototype tool in Smalltalk, and testing it on three different applications for six different data rates (ranging from very low to extremely high). Our results show that for applications that rely on a GUI, the time to build the GUI is reduced to 21 % of the original on the average.

3.3 Basic Observations, Assumptions and Restrictions

A first important observation is that code transmission over a network in general and more specifically in a wireless network is inherently slower than compilation and evaluation and this will remain the case for many years to come (section 2.2.4).

As a second observation we note that actual computer architectures provide separate processors for input/output (code loading) and main program evaluation and that in the near future we may expect more hardware support to employ parallelism (section 2.3.3).

We assume that the know-how and know-when of the migration of partitioned code is located in the sending host, so we apply a push strategy. However, this does not exclude the possibility of successful combinations with a pull-strategy but we did not implement this in our experiments.

We restrict us to the Smalltalk programming environment since it allows rapid prototyping, a high degree of reflection and it provides a much finer granularity than found in other programming environments.

The size of the applications and the data rate of the network are chosen under the restrictions imposed by current practical networks (see section 2.1.6 Window of Opportunity - page 39).

Compared with the approach of C. Krintz [Krintz et al. 1999] we propose a push technology instead of the code on demand pull technology and we divide our source code at the level of compilable units instead of at the class-level. Moreover, we not only simulate the wide range of transmission rates and ran our experiments in real time while C.Krintz et al. by their choice of programming environment (Java) were forced to restrain themselves to simulations.

Our experiments involve adapting and running real code and consequently our results are not obtained as part of some simulation technique. Only the different transmission rates are simulated in order to evaluate the technique on load channels ranging from very low to very high data rates. The results obtained for these simulated channels, with a constant data rate for all possible data sizes, can be adapted to a specific channel technology and protocol by taking in account the real data rate for the different data sizes. If for instance a TCP/IP network is considered, Figure 3 – page 38 can be applied to obtain these delays.

3.4 Profiling and Reordering

Before we can start to cut the code into different chunks we need to **permute** the source code in such a way that the code that will be evaluated first will be loaded first as well. After the cutting, we will need to apply some glue code. This glue code is needed to suspend the current evaluation of an application if the code to evaluate is not available yet. The detection of the presence of the code can be implemented at different levels. In languages that support reflection, each new method call can be forced to perform a reflective check to make sure all the resources are available but we choose for a more generic approach in the sense that it is applicable to all kind of programming languages. We add extra code at the end of each piece of code, implementing the function of **semaphores**. Semaphores will temporary suspend the application if the next chunk of code is not loaded yet.

To simplify the permutation process somewhat, during this first setup we assumed that the code flow is completely deterministic. In other words, we assumed that for each run of the code the application always behaves in the same way, hereby neglecting possible different user inputs or other real random events. This makes the permutation process straightforward since it suffices to determine the method invocation sequence once and rearranging the methods accordingly. The static structure of the permuted file will then reflect more closely

its dynamic behavior. Although this presumption may sound very harsh, we found that it was sufficient for the experiments we ran, especially with our focus on avoidance on user interface latency. The building of a user interface seemed to be a very predictable deterministic activity for the three applications in our experiments. All the evaluations of these applications showed the same method invocation sequence during the build process of the user interface.

Finding the ideal breakpoints is less straightforward. Profiling tools together with the dynamic behavior statistics, obtained as a side effect during the permutation process can give us some hints as where to split the code. In our experiments we will resort to some simple heuristics, such as cutting the file into four equal pieces. The permutation process, which is completely automated in our setup, consists of several distinct steps (Figure 13).

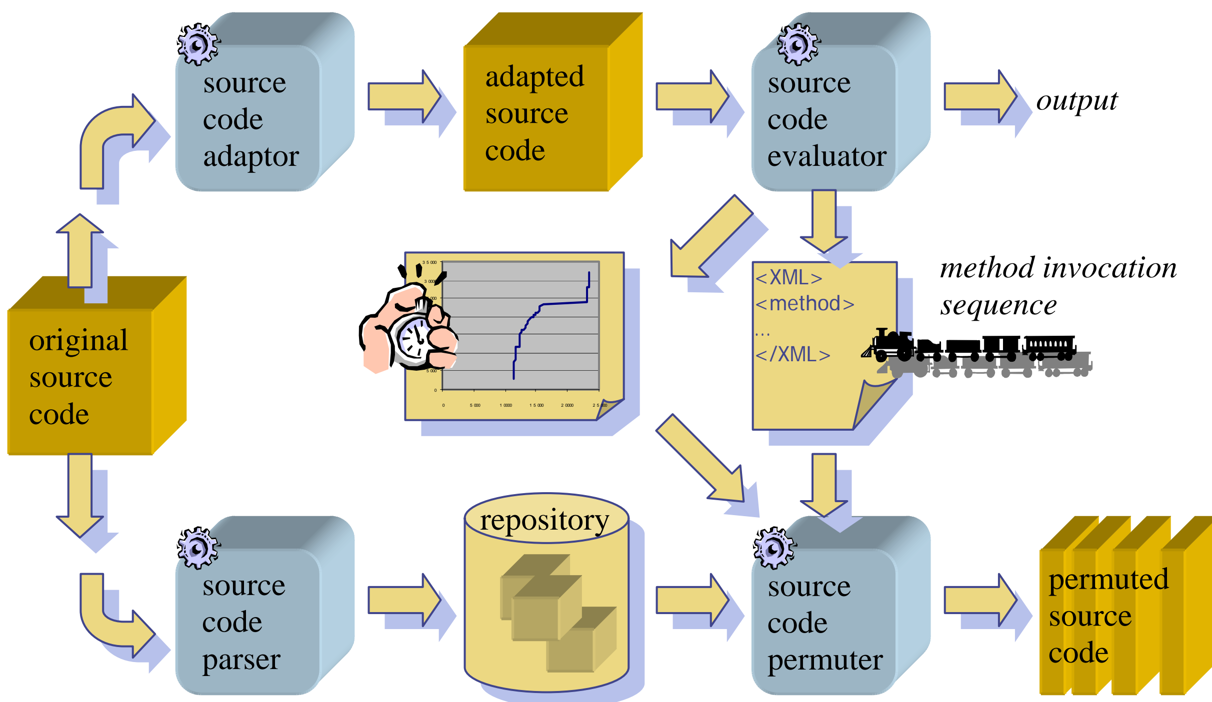

Figure 13: Permuting the source code

In order to obtain the necessary method invocation sequence the *original source code* is adapted (instrumented) with extra code that logs the time of invocation of each method. The instrumentation is accomplished by the *source code adaptor* component.

Then the *adapted source code* is evaluated. The output is ignored at this time but the instrumented methods will generate the necessary log information, in this case an XML[8] file that contains the *method invocation sequence*. As an additional output of the evaluator we also gather timing information that will serve as a guide to optimize the number of the different code pieces and the exact points to split up the pre-fetched source code.

[8] We chose for XML for two reasons. First, XML is a standard way for information exchange between different components. Second, the standard output format in VisualWorks 5i for Smalltalk source code is also XML.

In another phase, which can be carried out in parallel with the above steps, the *original source code* is parsed by the *source code parser* component and the resulting descriptions (class, methods, comments and other descriptions) are stored in an intermediate object *repository*.

In the final step a *source code permuter* will parse the XML file to retrieve the dynamic sequence of the method invocations and use this information to assemble the new *permuted source code* files that reflect this invocation sequence.

3.5 Reordering Algorithm

The algorithm used in this final step by the source code permuter is based on the dependencies between the different Smalltalk entities. A **method** cannot be loaded and compiled if the method's **class** description is not already available in the system. In the Smalltalk environment a method depends on its class description. So the class description will be written to the permuted source code file before the actual method.

In a dependency graph (Figure 14) this is depicted by an arrow from method to class. In the same spirit we notice that a class depends on its superclass, a class depends on its namespace, a class initialization method depends on its class description and depends possibly on semaphore code that eventually can prevent its invocation. A class also depends on the availability of relevant shared variables and, if the class is a subclass of `ApplicationModel`, the availability of the associated window specification resource.

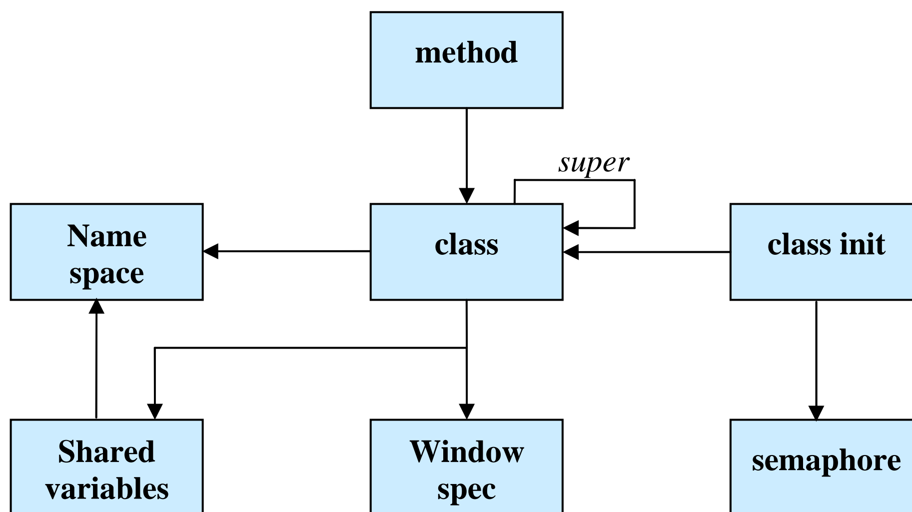

Figure 14: Smalltalk dependency graph

These dependencies are not complete so as to cover all the possible Smalltalk applications but are sufficiently comprehensive to cover all the dependencies in our actual experimental setup.

3.6 Pre-fetching

After the permutation process, we are now ready to pre-fetch the code. We propose progressive anticipative mobility using pre-fetching of permuted code as a technique that applies the idea of progressive transmission to software code instead of streaming sound or video.

Figure 15 shows the sequence diagram of a classic weak code loading process as known from applets that are fetched to a browser. When an external trigger launches the migration of code from Host 1 to Host 2 preparations are taken at Host 1 to load the code. For weak migration schemes this preparation is minimal since the code is not running yet and is mostly available, pre-packed, in a file data structure. Then the code is transported over the network and its evaluation started at the receiving host. The sequence diagram focuses on the handling of the code, i.e. the preparation, migration and evaluation of the code and ignores the assisting input/output processes that run on both host during the migration. To present the sequence diagrams in Figure 15 to Figure 17, the time to load the application and the time to compile and evaluate it are chosen to be the same. In reality this will depend of the size and type of the application and the data rate provided by the network.

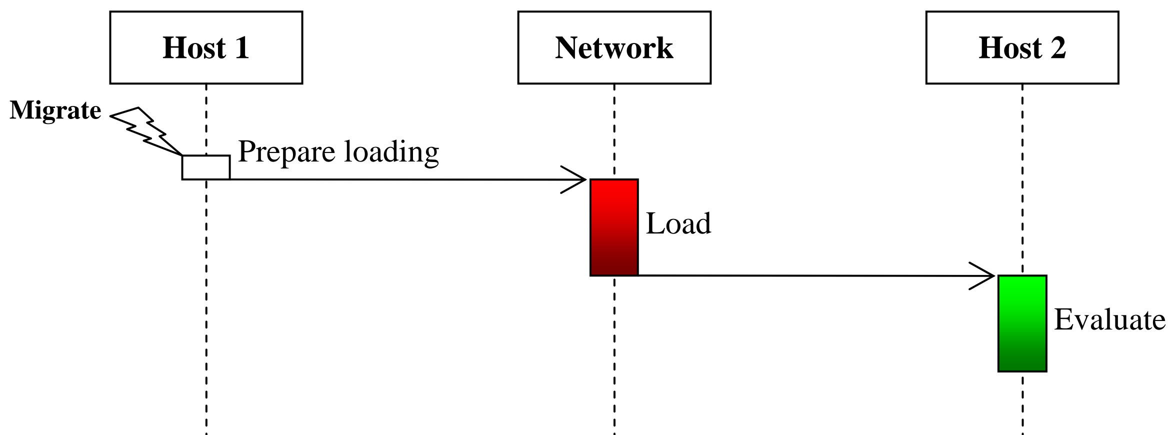

Figure 15: Normal weak code loading

Our proposed technique splits a code stream into several successive waves of code streams. When the first wave finishes loading at the target platform its evaluation immediately starts and runs in parallel with the loading of the second wave (Figure 16). The main difference with interlaced graphics such as progressive jpg is that we can use structural information about the code to determine the most ideal way of splitting the code into different waves.

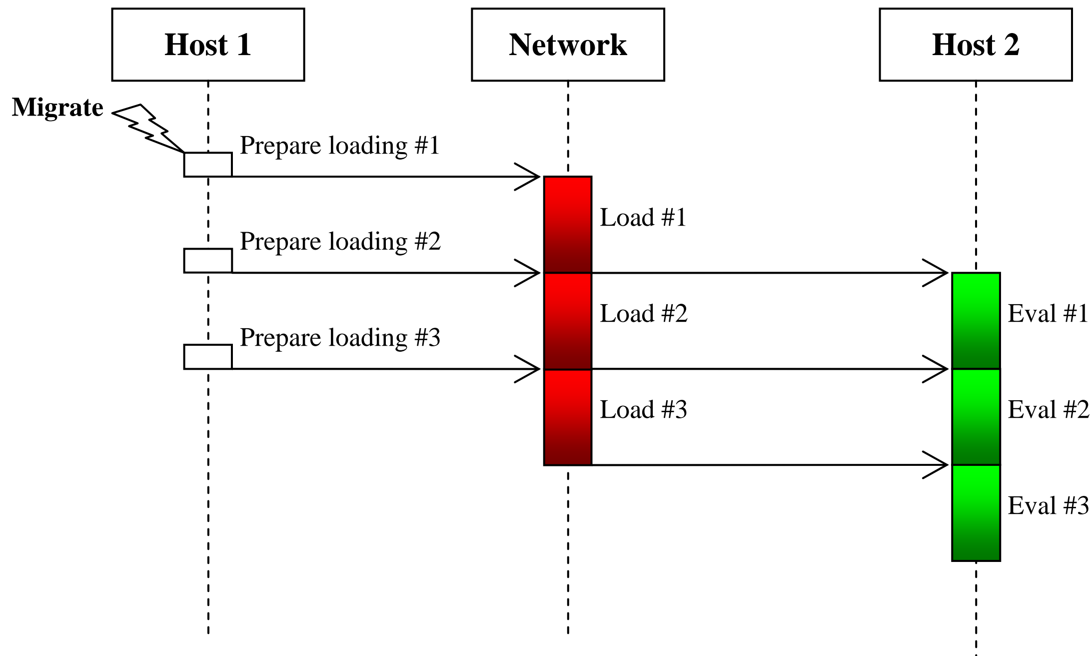

Figure 16: Progressive anticipative mobility using pre-fetching of permuted code

In a JIT compilation environment there is an extra compilation phase needed and therefore there are three processes that could potentially run in parallel: loading, compiling and evaluation (Figure 17). Extra time savings will only occur if different processors are deployed for the compilation and evaluation phase. Nevertheless, even if the same processor shares the processes of compilation and evaluation, the use of JIT compilation is advantageous for the proposed technique. Even code with a more complex flow of control, including system utilities and language processors such as optimizing compilers, written in C, are dominated by stable branches, and these branches usually vary little when the input data for the branch predictor changes [Fisher 1992]. Since the program flow of a classic compilation process is highly predictable, this guarantees that during this phase almost no unpredictable branches will occur, allowing a smooth parallel process between compilation and loading. In other words, incorporating a compilation phase in the evaluation flow of the program increases the predictable deterministic time zone at the start of that program.

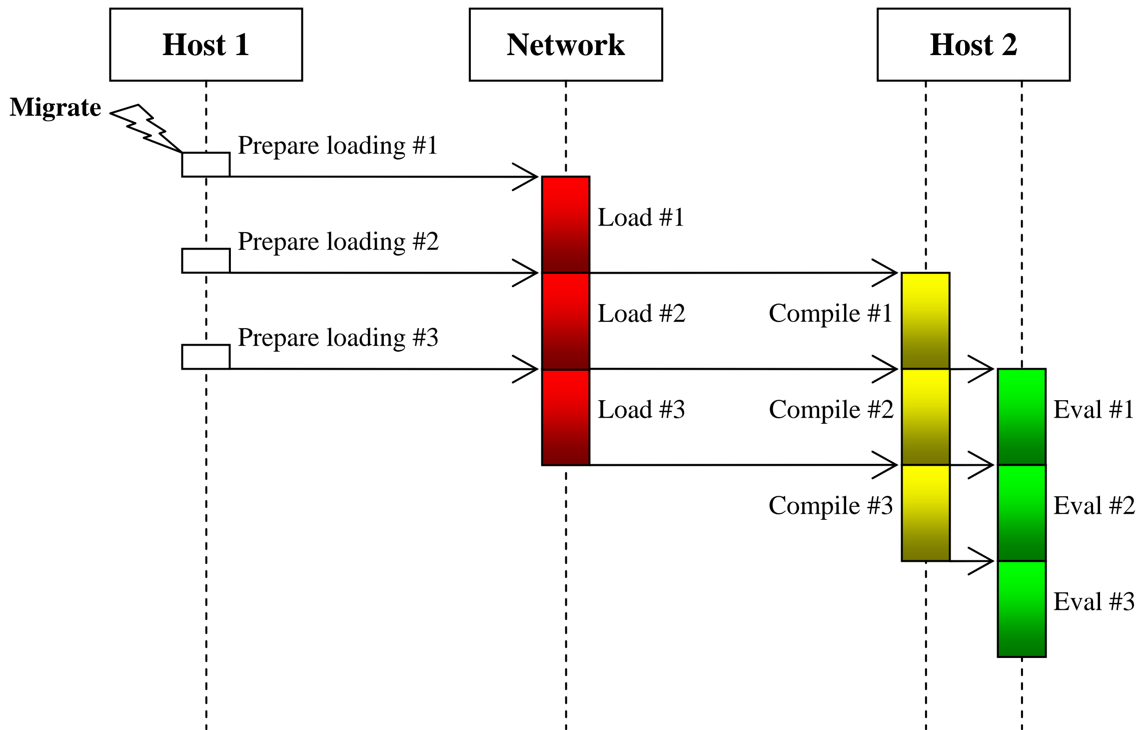

Figure 17: Pre-fetching of permuted code cycle in JIT environment

3.7 Experiment to Hide Network Latency

We describe some experiments to illustrate a generic approach of progressive anticipative mobility using pre-fetching of permuted code and to provide a proof of concept. A prototype tool was implemented in Smalltalk (more specifically, VisualWorks Release 5i.4), a popular object-oriented language that allows fast prototyping. We choose the granularity of the units of loading at the level of methods.

As a practical validation we tested our approach on three applications each exhibiting some typical but distinct behavior. It is not trivial to predict what type of applications will need to migrate in the upcoming ambient intelligent environment, therefore we tested three different applications with varying application size, number of classes, GUI size and use, and the utilization of independent threads or not. These applications come with the VisualWorks environment. They all have a size that is considerably greater than the minimum size of 2 kByte so that the transportation time is directly proportional with the size of the files. The maximum data rate applied is 42 Mbps, so below the limit of 1 Gbps by which, in networks over a great distance, the acknowledge protocol would dominate the transport time instead of the file size. See also section 2.1.6 : Window of Opportunity.

Benchmark: (ver: 5i.4) (80 kByte, 7 classes) An application program that runs different benchmarks on the Smalltalk environment adapted in such a way that after its Graphical User Interface (GUI) appears, it launches a standard test immediately, thereby simulating prompt user interaction.

CoolImage: (ver: 2.0.0 5i.2 with fixes) (184 kByte, 60 classes) an extended image editor that draws on a non-trivial graphical user interface.

Gremlin: (ver: Oct 7 '99) (65 kByte, 4 classes) An application that lets an animated figure pop up from time to time without the need for a user interaction, representing non-GUI applications that run as daemons in the background.

In order to test these applications we designed a code loader to simulate different transmission rates. Essentially the code loader waits for the amount of time needed to load the file containing the code, under different network data rates before effectively loading the code from disk and passing it on to the compiler:

```
[(Delay forMilliseconds: aFilename fileSize * 8 / self bps) wait]
```

This delay is proportional to the file size and makes abstraction of the underlying technology but provides a good estimation since, independent of the protocol, the input/output processors of the sending and receiving hosts as well as the network processors (routers etc…) will need an amount of time proportional to the number of bytes processed. To get the real delay for each technology (TCP/IP, GSM, GPRS …), the results can be fine-tuned to a specific channel technology and protocol by taking in account the real data rate for the different data sizes. If for instance a TCP/IP network is considered, Figure 3 can be applied to obtain these delays.

For this setup different transmission rates were simulated: 2400 bps (very low data rate), 14.4 kbps (slow modem), 56 kbps (fast modem), 114 kbps (GPRS) en 2 Mbps (UMTS). These different transmission rates were complemented by the rate obtained without network latency: 41 Mbps in our setup.

We deliberately chose for a JIT compilation approach because of its advantages in a low data rate environment: (1) Source code has a smaller footprint than the corresponding native code; (2) Source code preserves a high level of abstraction, thus enabling more powerful compression techniques; (3) JIT fits nicely in the proposed code pre-fetching technique since, as mentioned before, the compilation process is highly predictable, hereby increasing the deterministic time zone.

We evaluated our approach on the source code of our different applications: Benchmark, CoolImage and Gremlin. These applications each represent a more or less different kind of behavior.

First the source code of each application was automatically permuted using the different steps. For logging purposes a few extra lines of code were manually added to log the time the application needs to complete evaluation and also the time needed to produce its GUI or its first token of existence to the user. The application is loaded, compiled and run as is and then via a load channel simulating a number of different data rates, to gather the normal timing information referred to as "*normal end*" and "*normal GUI*" in the figures later. Next, the application is cut in four pieces. The following procedure is applied:

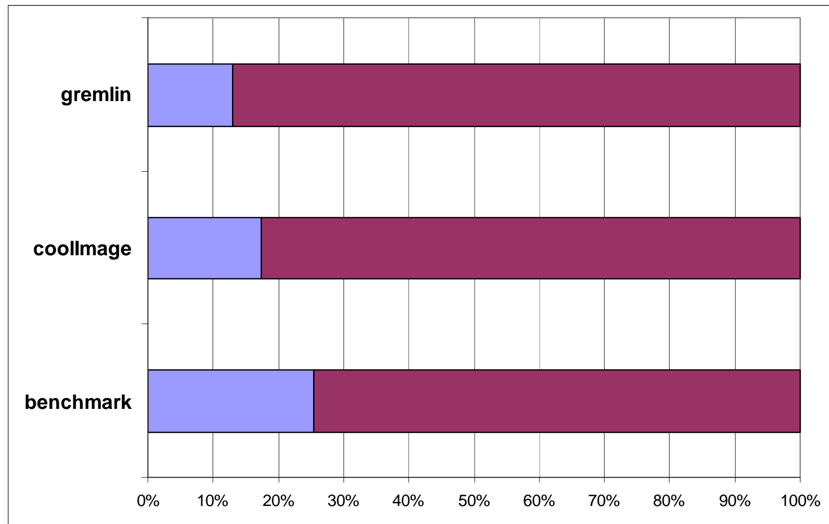

Figure 18: Percentage of code visited before the appearance of the graphical user interface

By examining the permuted code, it is fairly easy to determine the part of code visited by the evaluator in order to build the graphical user interface (Figure 18). For the user the emergence of the GUI is often the first indication that the underlying application is loaded and ready to go. In order to favor a quick emergence of the GUI we will try to make the first cut immediately after the GUI code. If the method that finishes the GUI is in the first half of the source code, as in our three test applications, then the first cutting place will be after that method. This will reduce the *user interface latency*, one of the main advantages of this approach.

As a result, the size of the first part is determined by the block of code visited by the evaluator in order to build the graphical user interface. The other parts are then constructed with, more or less, the same size as the first one. In the first experiment (Benchmark) the first part account for 25% of the total code, therefore the remaining code is equally divided in the three remaining parts: part 2, part 3 and part 4. We applied this approach to the other applications as well.

The exact cutting place will be just before or just after a method description in the files. In order to complete the dividing process, the four files need to get the same header and footer tags so as to make them valid XML files again, the required format of Smalltalk source code.

To prevent that code from part 1 calls code from part 2 before this part is arrived, a semaphore that halts evaluation until the code of part 2 is in place, is added at the end of part 1. The semaphore is placed at the start of the last method in part 1 to ensure that it will be encountered during the evaluation of that method. If the semaphore is placed at the end of the method it is possible that the evaluation of the method ends, as a result of a return statement, before the semaphore is encountered.

In this manner, three semaphores are added at the end of the three loose ends of part1, part2 and part3. The methods, in which the semaphores reside, are possibly invoked more than once during the evaluation of the application. This means that each semaphore must be disabled after its first use. In this setting this is done by enclosing each semaphore in a conditional structure in such a way that the semaphore is bypassed after its first use:

```
Pre-fetcher.S1Active ifTrue: [Pre-fetcher.S1 wait. Pre-fetcher.S1Active := false]
```

The application is then loaded, compiled and run again in a pre-fetching style for each of the simulated channel data rates and the new timing results are gathered. These are referred to as "*pre-fetched End*" and "*pre-fetched GUI*" in the figures later.

Each timing result is calculated as the average of three timing runs to be able to flatten occasional variations caused by the operating system or programming environment such as garbage collection.

3.8 Results

For each of the three test cases result times were measured with different data rates chosen under the restrictions imposed by current practical networks (see section 2.1.6 Window of Opportunity - page 39).

Six different data rates are simulated: 2400 bps (very low data rate), 14.4 kbps (slow modem), 56 kbps (fast modem), 114 kbps (GPRS), 2 Mbps (UMTS) and 41 Mbps (no network latency). For each of these data rates the time was measured in a normal set up (first load all the code and then compile and run) and a pre-fetched set up where the compilation and start of the code evaluation takes place after the first part is loaded.

For both loading types we measured the time it took for the GUI to display itself and the total time to complete the loading, compilation and evaluation of the application[9].

3.8.1 Benchmark

Benchmark is an application that runs selectable tests on the VisualWorks environment. In order to allow the code to run while the other parts are still transported, the application was adapted in such a way that after the GUI pops up the application immediately runs a number of standard tests. Figure 19 shows the parallel processes achieved for the lowest data rate of 2400 bps where the load times are significantly larger than the compile times. The figure also shows that although the code to build the GUI takes 25% of the total code (Figure 18), the time to evaluate this part is a fraction of the evaluation time of the remainder of the code. As a result the GUI will appear almost immediately after the loading and compilation of the first part of the source code. The application finishes before the last part has loaded. This indicates that the code in the last part was not needed in our setup and we may decide to stop loading the rest of the application.

Figure 20 and shows the behavior of the same application at increasing data rates. As the data rate increases to 114 kbps the load and compilation times become at the same order of magnitude. Here too, the application ends before loading completes.

Figure 21 shows the behavior with maximum data rate of 41 Mbps. It shows that at maximum data rate the load times are too short to take advantage of the parallel processing so in this case the evaluation process will end after the full loading and compilation process. A possible optimization as in the previous examples is not possible here.

GUI building indicates the first part of the evaluation process where the GUI is built. The second part of the evaluation process is indicated in de figure by *application*. The evaluation process has to share the processing power with the compile phases but can run in parallel with the load phases (except for load1).

[9] The experiments where carried out on a Dell® Inspiron 8100 computer with Intel® Pentium® III Mobile CPU AT/AT compatible processor at 1GHz processor speed and 256 Mb RAM running Windows® 2000 and VisualWorks 5i4.

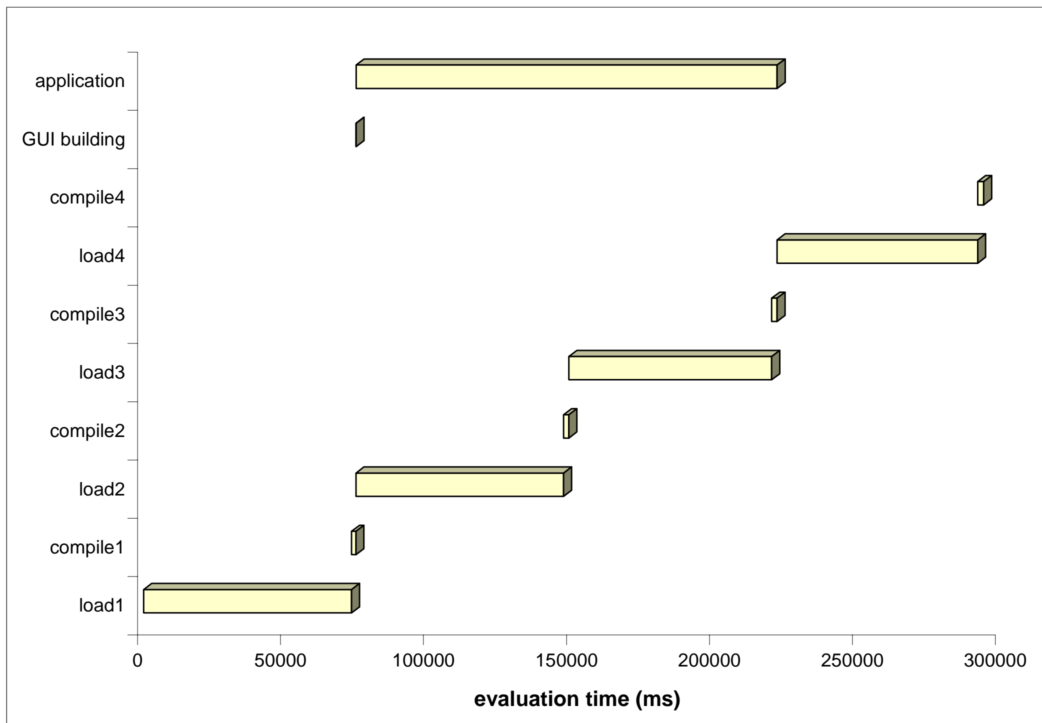

Figure 19: Parallel evaluation Benchmark @ 2400 bps

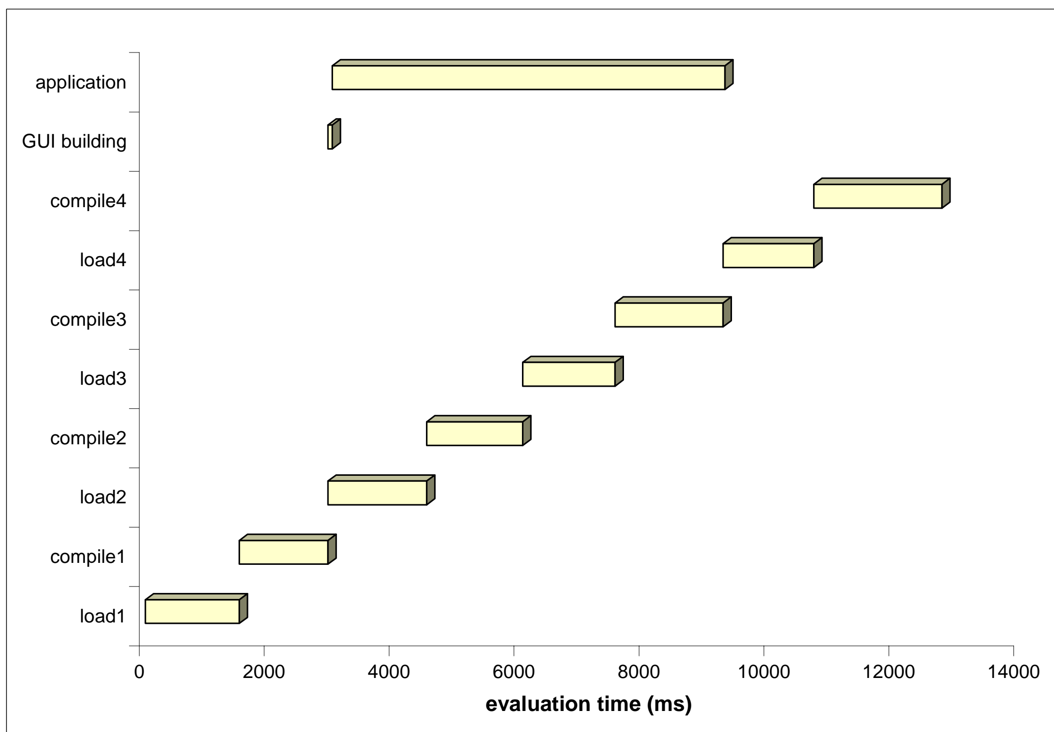

Figure 20: Parallel evaluation Benchmark @ 114 kbps

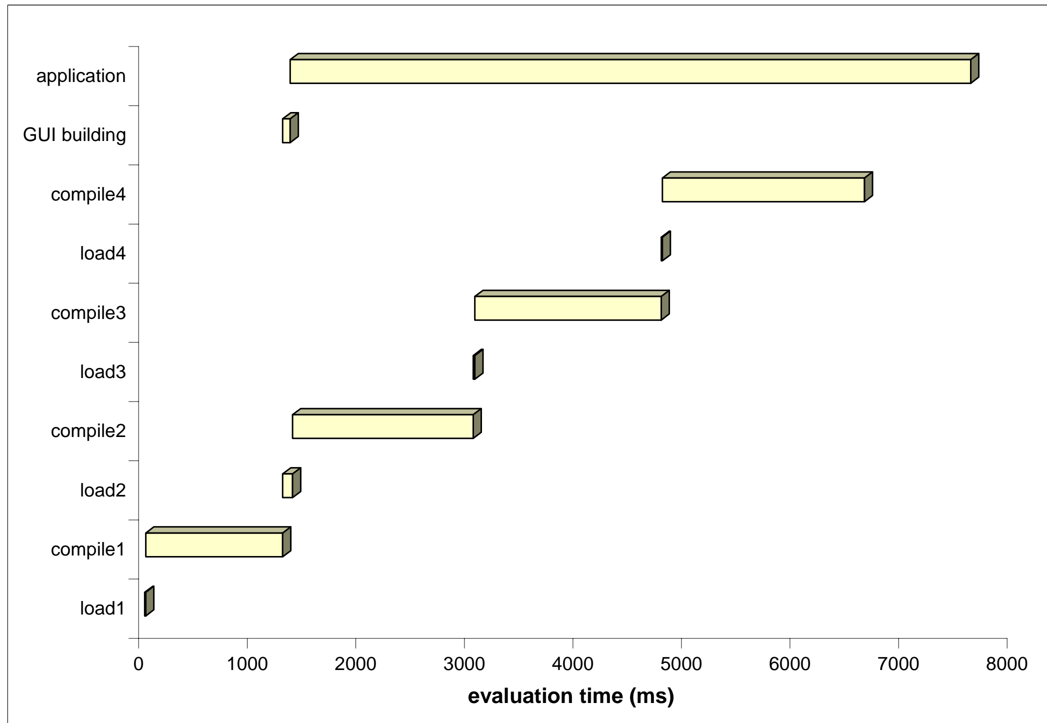

Figure 21: Parallel evaluation Benchmark @ 41 Mbps

Average timing results are depicted in Table 2 and Figure 22. Table 2 shows in the first row (*normal GUI*) the time in milliseconds it normally takes to render the GUI for the different data rates. The second row (*normal end*) shows the time in milliseconds the application normally needs to end. The third and fourth rows (*pre-fetched GUI* and *pre-fetched end*) show the same time if the application is deployed in a progressive anticipative mobility using pre-fetching of permuted code fashion. Finally the bottom rows (*GUI ratio* and *end ratio*) show the relative amount of time gained by pre-fetching to present the GUI and to finish the application. Table 3 shows the standard deviation of the different timing results to give an indication of the average deviation.

Table 2: Average timing results (ms) for Benchmark application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	279268	51184	18074	12352	7016	6562
normal end	280255	52175	19069	13361	8133	7526
pre-fetched GUI	74327	13341	4669	2995	1722	1341
pre-fetched end	221564	40291	14279	9279	6062	7609
GUI ratio	26,61%	26,06%	25,83%	24,25%	24,54%	20,43%
end ratio	79,06%	77,22%	74,88%	69,44%	74,54%	101,10%

Table 3: Standard deviation (ms) of timing results for Benchmark application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	548	235	566	744	529	309
normal end	555	239	566	739	690	311
pre-fetched GUI	268	186	9	192	355	37
pre-fetched end	1083	680	596	276	631	482

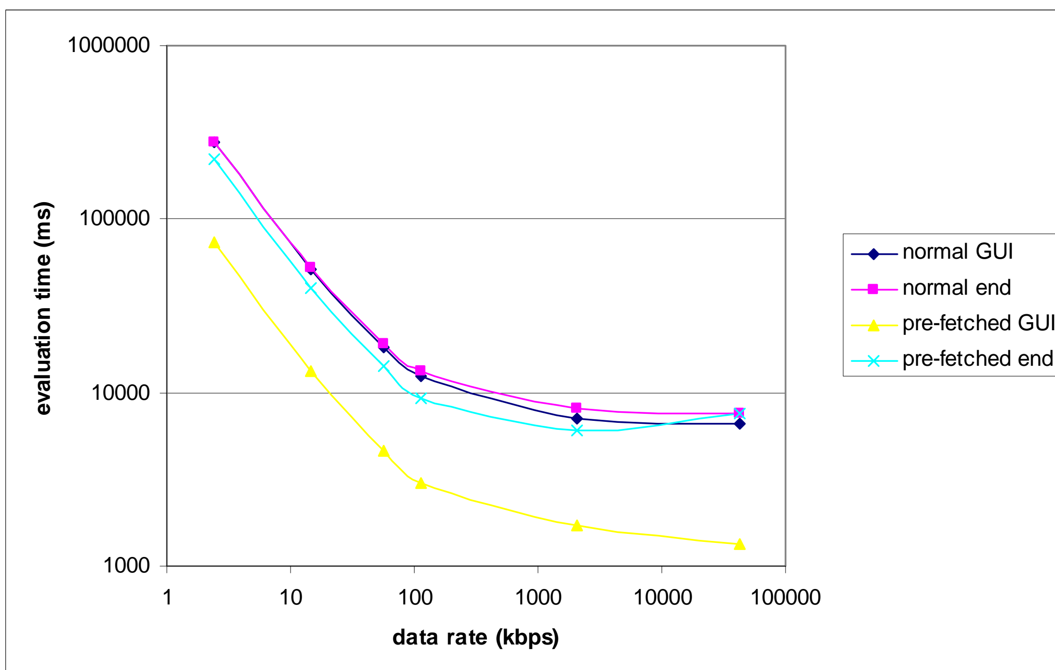

Figure 22: Average timing results for Benchmark application

Figure 22 puts the results of Table 2 in a graphical view. Note that the x and y scale are logarithmic to accommodate the wide range of data rates. Note also from Figure 22 that, if the application is loaded via a network, the application itself ends earlier (on average 75% of the original time needed) if deployed in a pre-fetched mode. This is possible because the evaluation of the application already starts after the load and compilation phase of part 1 and does not have to wait until the complete source code is loaded and compiled.

3.8.2 CoolImage

CoolImage is the largest application of the three which generates a large GUI and then waits for user interaction to draw icons. As a result, the end of the loading and compile phase is practically the same for the pre-fetched and normal deployment simply because in this experiment no action takes place after the GUI building.

Table 4: Average timing results (ms) for CoolImage application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	640540	114905	38978	25348	13624	12666
normal end	640545	114909	38982	25351	13628	12673
pre-fetched GUI	115005	22832	8847	6663	4780	4224
pre-fetched end	638613	115496	39135	25192	13888	13037
GUI ratio	17,95%	19,87%	22,70%	26,29%	35,09%	33,35%
end ratio	99,70%	100,51%	100,39%	99,37%	101,91%	102,87%

Table 5: Standard deviation (ms) of timing results for CoolImage application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	131	478	166	166	584	454
normal end	132	479	165	166	585	452
pre-fetched GUI	351	400	152	221	320	155
pre-fetched end	536	305	180	156	82	392

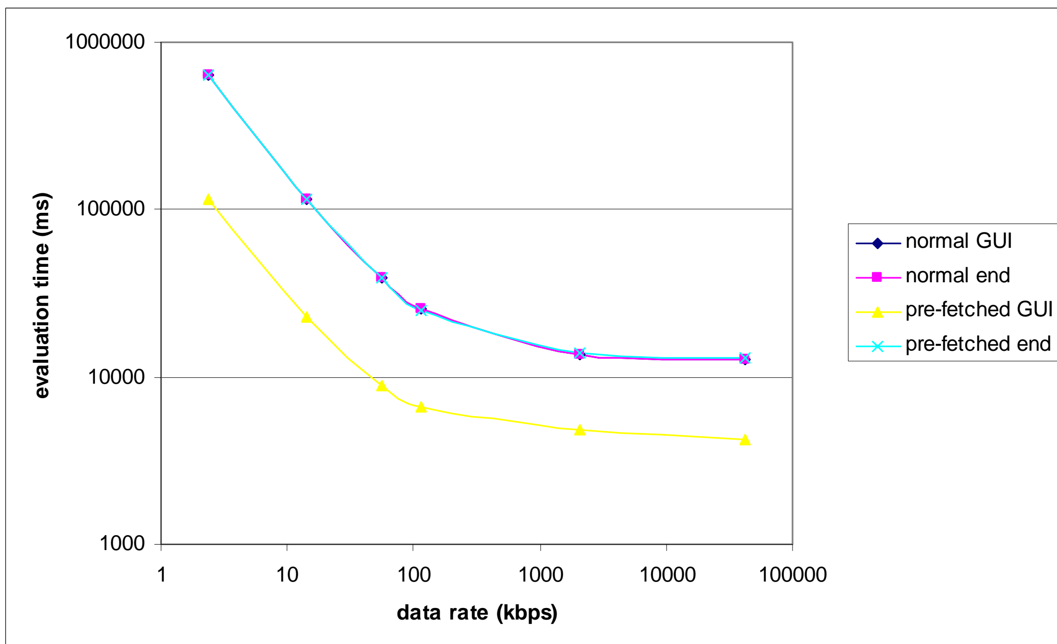

Figure 23: Timing results for CoolImage application

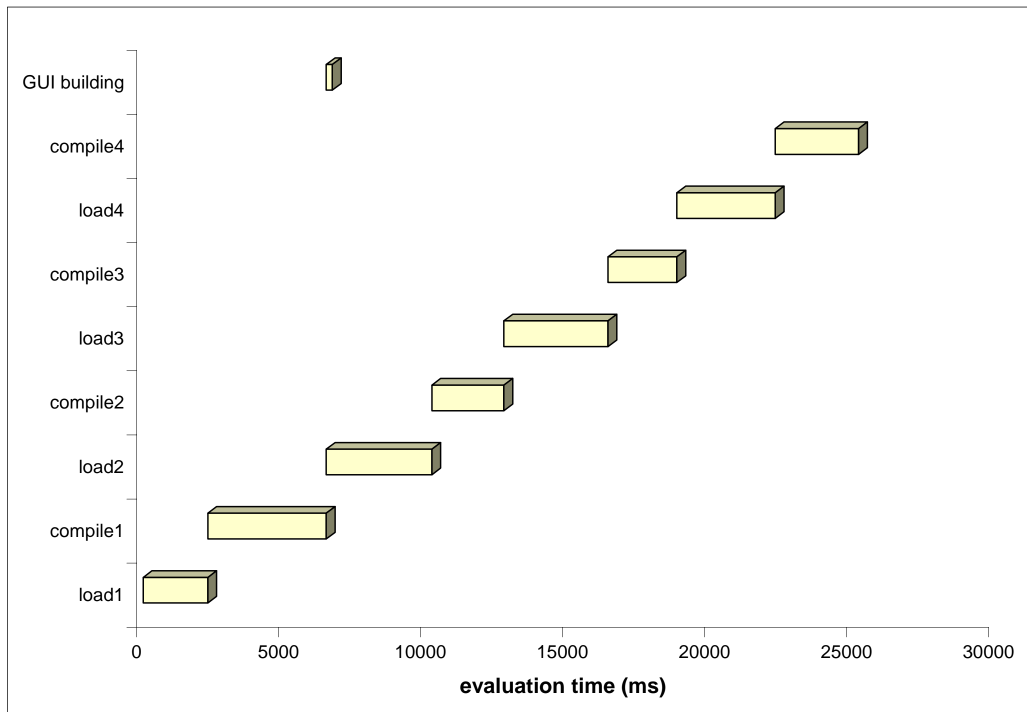

Figure 24: Parallel evaluation CoolImage @ 114 kbps

Figure 24 shows the parallel processes achieved for a data rate of 114 kbps. Note the difference with Figure 20 were after the GUI building the application continued to evaluate some code. In this type of application the application stops and waits for user input after displaying the GUI. It is also clearly visible in Table 4 and Figure 23 that the time the application itself will needs to end will not vary. In this case this is the time to load and compile all the methods that can be invoked via the GUI.

3.8.3 Gremlin

Gremlin is an application that runs in the background of the VisualWorks environment and pops up an animated figure from time to time at the border of the active window. When the application is launched, the animated figure pops up for the first time and a help window shows up. Table 6 and Figure 25 show the delays of the Gremlin application.

Since the Gremlin application starts with a popup of an animated figure and during the rest of its life it just does the same over and over again at different time intervals it means that all the resources need to be in place before the application can start. This is reflected in Figure 25 by the fact that only for data rates lower than 56 kbps the first popup can finish earlier than the complete loading and compile process. For data rates greater than 56 kbps it is the popup process itself that will determine the end of the process. The appearance of the GUI in the pre-fetched deployment for data rates lower than 56 kbps however is much faster and in the same order as the other tests (on average 25% of the original time needed).

Table 6: Average timing results (ms) for Gremlin application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	230743	46392	19563	14713	10469	10262
normal end	230745	46394	19565	14715	10471	10264
pre-fetched GUI	51601	15194	11932	11005	10283	10439
pre-fetched end	225385	39441	12777	11005	10283	10439
GUI ratio	22,36%	32,75%	60,99%	74,80%	98,23%	101,72%
end ratio	97,68%	85,01%	65,31%	74,79%	98,21%	101,70%

Table 7: Standard deviation (ms) of timing results for Gremlin application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	302	326	173	258	285	407
normal end	302	326	173	258	285	407
pre-fetched GUI	162	209	293	233	206	218
pre-fetched end	851	322	326	233	206	218

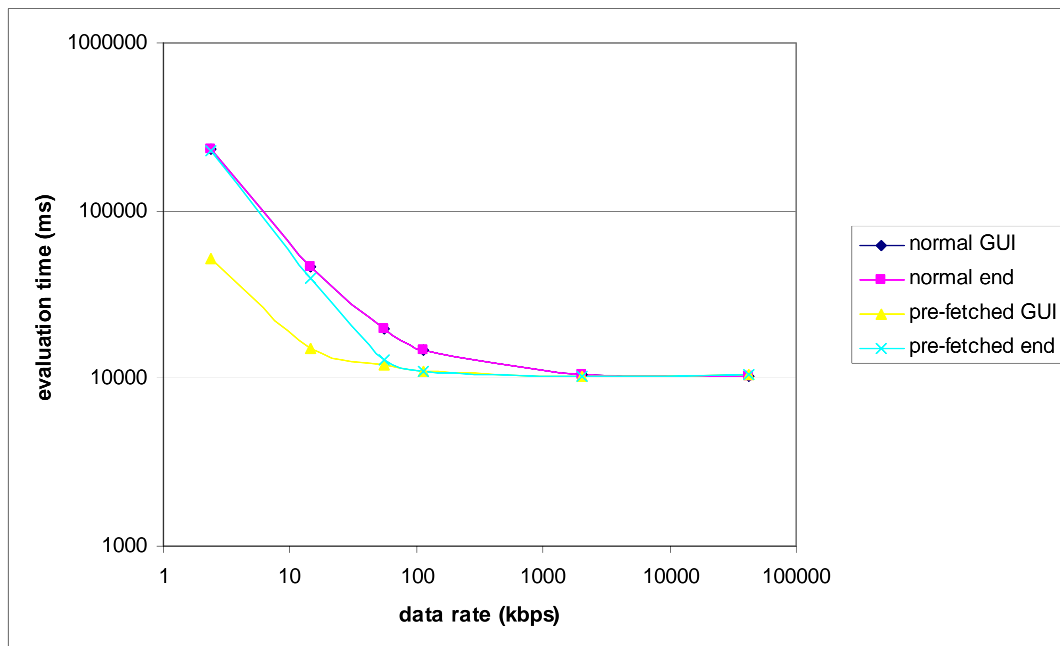

Figure 25: Timing results for Gremlin application

3.8.4 Adapted Gremlin

The poor results obtained with the Gremlin application led us to the question whether it is possible to adapt the design of the application in such a way that pre-fetching could be applied more advantageously. If we could change the application so that it would no longer depend on all of its resources, for its first sign of life, this would do the trick.

Table 8: Average timing results (ms) for adapted Gremlin application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	223468	39420	12568	7890	3544	3150
normal end	223470	39422	12569	7892	3546	3152
pre-fetched GUI	44183	8261	3049	2223	1413	1228
pre-fetched end	224902	39441	12596	7968	3557	3197
GUI ratio	19,77%	20,96%	24,26%	28,17%	39,88%	38,98%
end ratio	100,64%	100,05%	100,22%	100,97%	100,29%	101,45%

Table 9: Standard deviation (ms) of timing results for adapted Gremlin application

data rate (kbps)	2,4	14,4	56	114	2048	42308
normal GUI	373	409	401	430	394	226
normal end	374	409	401	431	394	226
pre-fetched GUI	70	48	24	106	150	37
pre-fetched end	500	246	217	191	102	70

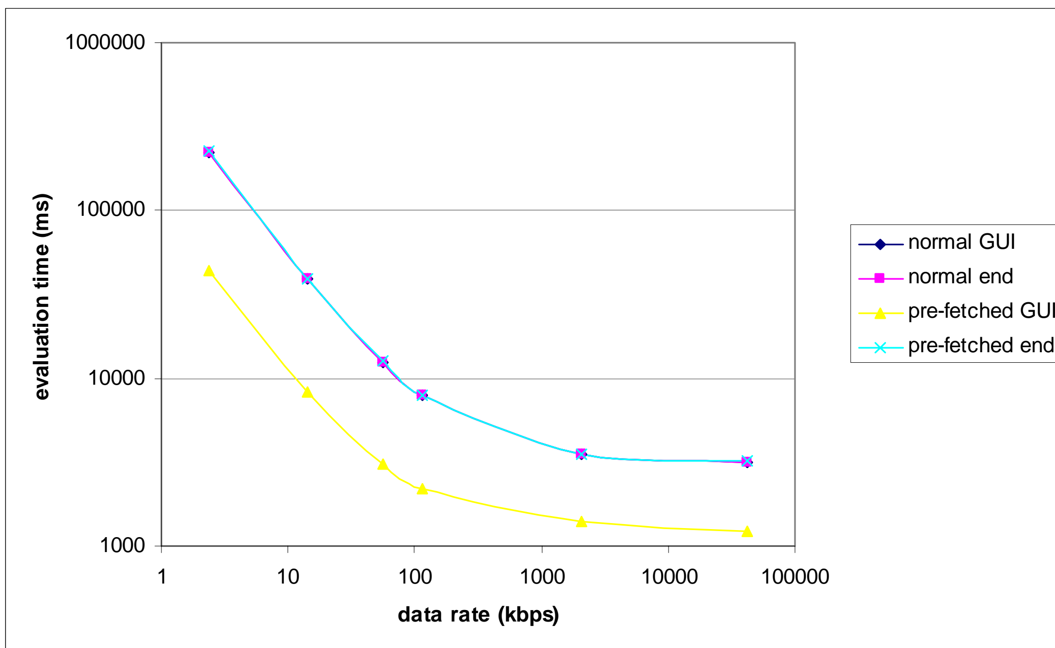

Figure 26: Timing results for adapted Gremlin application

To achieve this, we adapted the Gremlin application so that after it is launched only the help window appears (containing an explanation of the behavior of Gremlin and stating that the first popup is scheduled within 5 minutes). Conceptually this does not really change the main behavior of the application but as Table 8 and Figure 26 shows there is now a significant time gain possible for the GUI building (now the text window) and the end of the application (now the loading and compilation of the source code but before the first popup).

Apparently small changes on the design level of the application sometimes suffice to get a more optimal behavior in a pre-fetching loading environment. High-level analysis is required however for this kind of optimization because the resemblance of the functionality of the two versions of Gremlin becomes only apparent at the level of the user perception.

3.9 Discussion

3.9.1 Speedup

If the graphs *pre-fetched end* and *pre-fetched GUI* appear **below** the *normal end* and *GUI end* graphs a speedup was achieved. Note that Figure 22 is the only diagram where none of the graphs coincide with each other. This is because the Benchmark application is the only example that runs some time consuming benchmark tests after the appearance of the GUI. The two other applications wait for user interaction after building the GUI.

3.9.2 Application Speedup versus Data rate

Figure 27 shows the relative amount of time needed to present the GUI compared with a normal non-pre-fetched setup for the different data rates. If we neglect the original non-adapted Gremlin application we find that an average speedup of 25% is obtained.

For applications were the GUI building takes a relatively large part (such as CoolImage and Gremlin) the speedup achieved by pre-fetching seems to decrease as loading speed increases. In the extreme case of Gremlin where the GUI building needs all the resources in place, the application takes even a slightly longer time to evaluate. This is because the extra semaphore code in the source code and the code to guide the pre-fetched loading process yield an extra overhead, and are responsible for time ratios higher than 100 %.

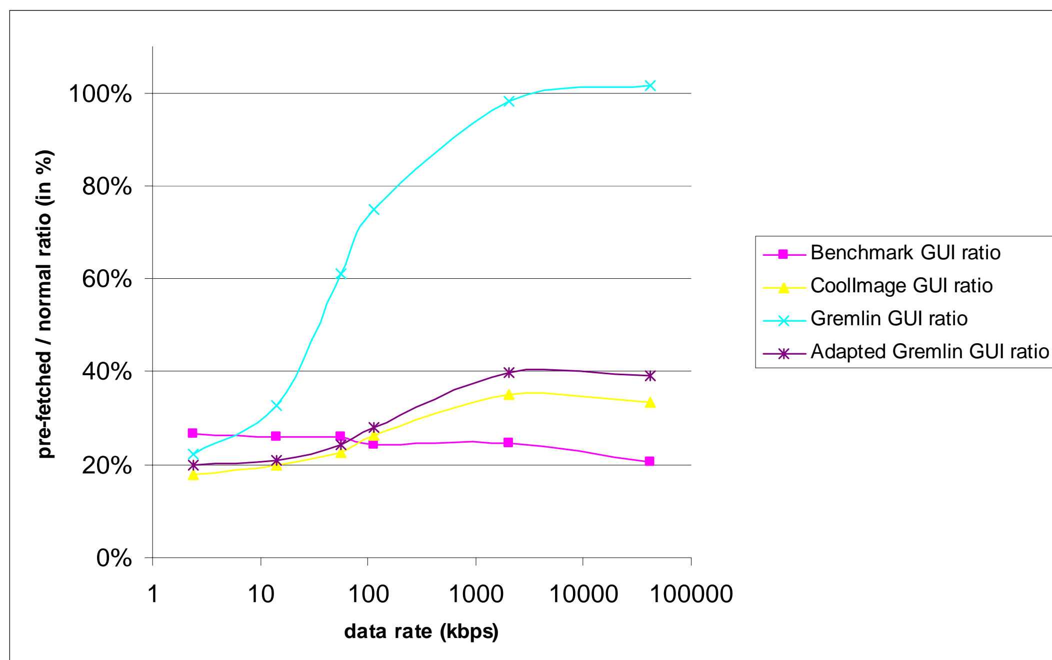

Figure 27: Time needed to build GUI compared with original time

Figure 28 shows the relative amount of time needed to end the application for the different data rates. The total time to load and evaluate the application at maximum data rate will be the same or slightly higher due the extra source code added. The dip in the original Gremlin graph indicates that for data rates smaller than 56 kbps the total time is determined by the load and compile processes but for data rates greater than 56 kbps it is the popup process that will determine the total time needed. Therefore, the right part of the figure the graph will look identical to the one in Figure 27.

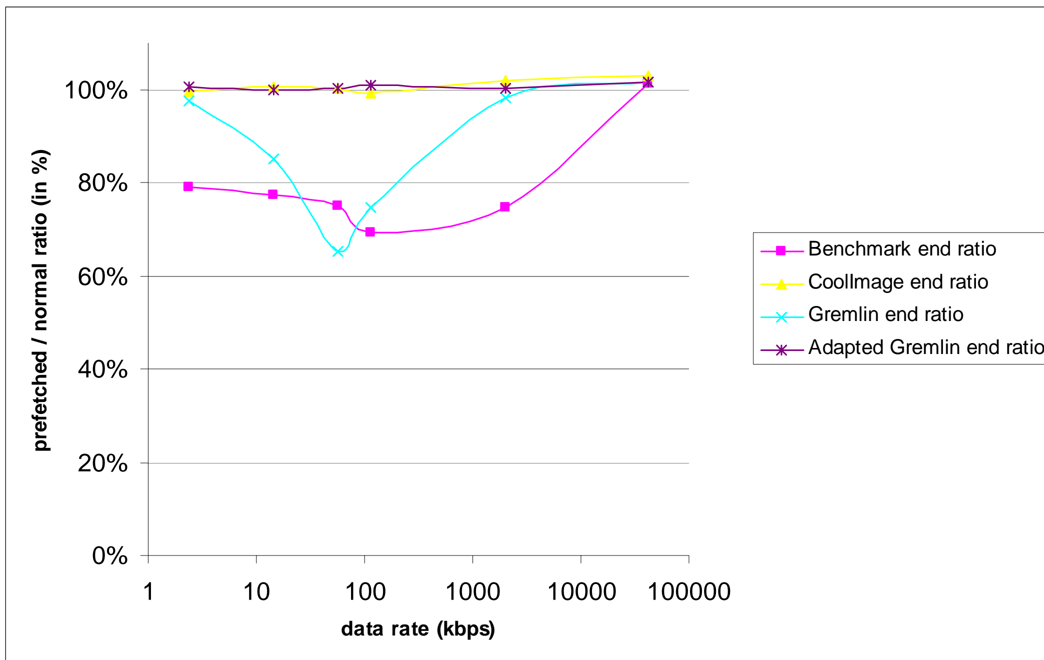

Figure 28: Time needed to end the application compared with the original time

3.9.3 Pre-fetching Guidelines

As became apparent in the Gremlin case it can be advantageous to adapt existing programs to make full use of the power of pre-fetched loading. Especially when you write new applications from scratch it is possible to keep in mind some guidelines that will lead to an optimal progressive anticipative mobility using pre-fetching of permuted code. Some of the obvious ones are:

- Keep programming modules independent from each other (i.e., use low coupling and high cohesion).
- Start as soon as possible with building the GUI.
- Keep the code and the resources to present the first user interface as small as possible. Mostly this is the GUI where the user is confronted with at startup.
- If necessary enhance the GUI, e.g. extend the GUI menu, at a later time.
- Postpone heavily resource-dependent actions as long as possible.
- Postpone multithreaded processes as long as possible.

3.9.4 Dealing with Semaphores

As mentioned before, precautions must be taken to prevent methods from being called that are not loaded yet. Although it is possible to catch these exceptions on the level of the virtual machine or even on the level of the operating system, for this setup we chose for the approach of adding semaphores in the source code since this provides a very generic mechanism that can be applied in many programming environments.

It can be assumed that for every application there will exist an ideal number of pieces to split the code in so as to obtain a maximum speedup. If the number of pieces increases so will the total size of the code since each piece of code will need extra statements to present the semaphore code. And if the code size increases so will the loading time and since the extra code needs to be evaluated too, so will the evaluation time. These are just the times that we wanted to decrease in the first place. Furthermore, there will be an extra overhead at the

receiver and sender platform to administrate the loading, compiling and evaluation of the different parts.

Provisions need to be made to disable the semaphores once they have served their purpose for the first time. Placing them in a conditional branch that bypasses them after first use seems to be a valid option and this is the choice that we took in our experiments.

If the method in which the semaphore is placed is triggered a significant number of times, complete removal of the semaphore code after its first use can be considered. Access to a precompiled version of the same method without the semaphore code can speed up that process.

Another possible approach is deploying a dedicated kind of garbage collection agents to remove unused semaphores in the background. On the other hand, if we are dealing with mobile code that moves continuously from one host to another it may be advantageous to keep the semaphores in place.

Semaphores will always have a negative influence on the performance of an application so caution should be taken for time-critical systems.

3.9.5 Applicability in other Environments

Smalltalk is a language that is interpreted by a virtual machine and is available on a wide variety of platforms. Therefore we expect that the general behavior of our experiments will be the same on different platforms such as MacOS or UNIX. We might expect some differences however for example related to character presentation in files and low-level window redraw events.

Java is also a language interpreted by a virtual machine available on different platforms. The security model of Java however prohibits the segmentation of the code in methods. In Java, the unit of code loading is the class. A class needs to be loaded entirely to allow the security mechanisms to calculate his signature. Krintz et al. [Krintz et al. 1998, Krintz et al. 1999] ran simulation test on Java code and obtained similar results. Our results shows that even for environments without network latency the user interface latency can be diminished (31% on average) simply by running the disk-load and compile process in parallel with the evaluation of the first part of the program. The methodology proves to be very generic and applicable to all systems where code needs to be moved before it is evaluated.

Experiments on interleaved file transfer [Krintz et al. 1998] (section 2.3.6.3 page 58) yielded comparable results. Transfer delay could be decreased between 31% and 56%. An important difference with our approach is the implementation language (Java instead of Smalltalk). Moreover, because of the limitations of the Java virtual machine security model, Krintz et al. simulated their experiments using a bytecode instrumentation tool called BIT [Lee 1997]. Additionally, they only considered two different data rates while we explored a wider range of 6 different data rates. Alternatively, they proposed to transfer different pieces of Java code in parallel, so as to ensure that the entire available data rate is exploited.

3.10 Summary and Conclusion

In order to support the inherent dynamics in an ambient intelligent network we will need not only to send data but we will need to send behavior (code) also. Since the width of the timeframe available to migrate the code is not predictable, we need some kind of mechanism to break up code into smaller parts and send them one by one, progressively in time, to the receiver. This will increase the possibility that they will fit in the temporal timeframe.

Precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately usable (ready for evaluation) at the receiver's end.

In this chapter we proposed a technique that breaks up code in smaller parts and applies the idea of progressive transmission to migrate software code. We described the technique of code permutation. A technique to permute Smalltalk source code at the level of compilable units based on the dependency graph was presented. A prototype tool in Smalltalk was build that automatically permutes a Smalltalk source file and generates a set of source code files optimized for the pre-fetching process.

Performance of an application is most commonly measured by *overall program evaluation time* and network performance is most commonly measured in *network latency* but in a mobile environment performance is also measured by *application availability, invocation latency*, and *user interface latency*.

Overall program evaluation time is the time between the invocation of an application and the end of the evaluation of the last instruction.

Application availability is the inverse of the time an application "freezes" during migration.

Network latency is the time the application needs to travel over the network.

Invocation latency is the time from application invocation to when evaluation of the program actually begins.

User interface latency is the time a user has to wait between his demand and a user interface reaction of the system.

Table 10 gives an indication of the performance of the presented technique in these different domains. If the technique has a small advantage in a particular domain it is indicated with a +, an advantage in a domain is indicated by ++ and if the technique excels for a particular domain it is indicated with +++. Note that the table only lists the positive performance properties. Other properties that may have a negative influence by the introduction of progressive mobility as the total size of the code, development overhead, maintainability, extra security issues etc… are not considered in this context and are left for future work.

Table 10: Properties of the Pre-fetching Technique

	Overall program evaluation time	Application availability	Network latency	Invocation latency	User interface latency
Pre-fetching of permuted code	++	+	+++	++	+++

To conclude this chapter we discuss these results in the most important dimensions of the conceptual framework provided in chapter 2.

3.10.1 Network

We optimized the experiments to reduce network latency in general and user interface latency in particular, so it will come as no surprise that these are the domains in which the results excel. Exploiting parallelism between loading and evaluation proves to reduce user interface latency considerably (21% of the original time on average in three applications tested). The overall program evaluation time decreases since not all the code has to be transported and compiled. The overall program evaluation time can be significantly reduced (79% of the original time on average in three applications tested).

If the user interface latency is reduced, the invocation latency is reduced too but since we optimized the technique for user interface latency reduction at the cost of delaying the invocation of the other processes of the application the score of invocation latency is less.

Latency is an important dimension in networks but it is important to keep in mind that also in an ambient intelligence environment two kind of networks must be considered, connection-oriented networks that will need an extra setup time for each exchange of data and connectionless networks that can exchange block of data much faster.

If the setup time in **connection-oriented networks** is very large compared to the time to send one of the parts in which we divided the application it would make no sense to send them one by one progressive in time and pay for each transmission the extra setup time.

In this case, it would be better to use a classic migration scheme and transport the complete application at a whole so that only one setup time is needed.

However if the setup time is of the same order of magnitude as the time to send one of the parts and the network architecture allows to keep the connection open after the first part is migrated then we can still apply the pre-fetching technique after the extra setup delay.

Connectionless Networks are the ideal environment to implement progressive anticipative mobility using pre-fetching of permuted code. Compared to the connection-oriented networks the packets are typically a little larger since they need to contain the complete address of the receiver but in practice the size of this extra data can be ignored compared to the size of the actual block of data sent.

3.10.2 Application

The feasibility of the pre-fetching technique has been validated by implementing prototype tools in Smalltalk. As a practical validation we tested our approach on three applications each exhibiting some typical but distinct behavior, with varying application size, number of classes, GUI size, typical usage, and the utilization of independent threads or not. Our results show that for our test applications that rely on a GUI, the time to build the GUI is reduced to 21 % of the original on the average.

Application availability is not really an issue here since the application is not running at migration time but since it becomes available much faster than by using classic migration scenarios we give it a small positive indication in Table 10.

3.10.3 Techniques

The most relevant technique applied here is the exploitation of parallel processing of the migration of the code and the evaluation of the code on the receiver.

Complete anticipative mobility using pre-fetching is only possible if the program flow is known in advance. If the program flow is not deterministic it still remains possible to permute the source code to reflect the most probable path to optimize parallelism as much as possible. Several techniques are developed to find out this most probable path [Jason and Patterson 1995].

In this chapter we did restrain ourselves to weak mobility; we will handle progressively strong mobility in the next sessions 4 and 5.

4 Progressive Mobility using Component Streams

Speed is good only when wisdom leads the way
-- James Poe (1921–1980)

4.1 Abstract

Networks in ambient intelligence environments are more volatile than static networks. In order to support this new network architecture where connections between partners are no longer predictable and where the connection time may be less than a second there is a new need for techniques to exchange information as fast as possible. In ambient intelligence environments the information exchanged will sometimes take the form of code in order to provide support for the systems dynamics inherent in these new environments.

During the migration of this code the application itself is typically not available to interact with other processes, which in some cases might be not acceptable, so there is also the need for mechanisms that allows the code to continue its evaluation during the progressive migration so that the application remains available for users or other applications at all time.

Progressive anticipative mobility using pre-fetching of permuted code migrates static code, code that is not started yet in a progressive mode.

Progressive mobility using component streams takes this approach a step further by progressively migrating running code.

Progressive mobility using component streams allows applications to migrate from host to host without sacrificing evaluation time during the migration phase and it hides network latency since it also allows the application to start at the receiving host much earlier. Progressive mobility using component streams goes beyond strong migration since the evaluation of the application will never be halted.

In this chapter, where we provide a proof of concept of the exploitation of components streams, we start by describing the proposed technique and the different migration strategies that follow from it. We describe the different experiments conducted on this theme and provide design guidelines to optimize applications that need to migrate by components streams.

Roadmap:

- Introduction
- Proposed Technique
  - Basic Observations, assumptions and restrictions
  - Technique description
    - Compensating Network Latency
  - Migration Strategies
    - Self Triggered after Last Instruction
    - Self Triggered based on Profiling
    - Under Control of a Supervisor
    - Fixed Migration Strategy
    - Dynamic Migration Strategy
  - Discussion
- Experiments
  - Experiment to hide network latency
    - Borg environment
    - Java environment
  - Experiment to reduce system latency in low data rate environment in the Java environment
  - Experiment to reduce system latency by parallel component evaluation in Smalltalk environment
- Design Guidelines
- Summary and Conclusion

4.2 Introduction

Ubiquitous Communication in ambient intelligence environments only makes sense if the objects that compose the network are available to respond to the requests of other objects.

Besides the exchange of data between the everyday objects in an intelligent environment we will need to send also code in order to address some key issues to support the dynamics of the system. It will not always be possible to migrate this code as a whole, since the connection time between, possible moving objects, is not predictable.

Connections between hosts in these new environments are more volatile than in static networks, so there is the need for mechanisms to split up the code in smaller parts that will fit in the limited timeframes in order to migrate the parts of the code progressive in time to other objects.

To keep the application available during its migration we will also need a system that allows the code to continue its evaluation during the progressive migration so that the application remains available for users and other applications at all time.

In this second theme, it is our goal to build a proof of concept of a system that breaks up the code of an application in smaller parts and sends them one by one progressively in time, while the evaluation of the application continue. Since the evaluation should continue during the migration we will explore this theme preferably in a setting that supports strong mobility.

We will explore three different possibilities to harness parallelism. We will exploit the same kind of parallelism as in the previous chapter, starting the evaluation of parts already received while the rest of the application still needs to migrate. But since the application is already running before its migration we will also exploit the parallelism of the migration of copies of components that still are running on the sender and the parallelism between components still running on the sender and components already arrived and running on the receiver.

The feasibility of the technique has been validated by implementing prototype tools in the Borg mobile agent environment and later also in Java and Smalltalk. Our experiments show that this migration strategy can hide network latency almost completely.

In this second theme, our contribution is the introduction of progressive mobility using component streams.

The main characteristic of transmission schemes as audio and video streaming is that the processing of the digital stream is started long before the load phase is completed.

The newly introduced term progressive mobility using component streams is inspired by streaming media but also by the transport mechanism for a sequential file, a data structure that allows only sequential access. During the streaming process, the first part of the file will be already located at the receiving host while the other part of the file still remains on the sender platform. When streaming a running application, part of the application will already run on the receiving host while another part is still running on the sending host.

Progressive anticipative mobility using pre-fetching of permuted code also applies a technique where code arrives and starts its evaluation on the receiving host computer before the load phase is completed, but the main difference with component streams is that the technique of pre-fetching migrates code from an application that is not running yet. With progressive mobility using component streams we migrate running code. This kind of migration is known as *strong mobility* while the former is called *weak mobility* [Fuggetta et al. 1998].

Strong and weak mobility only differ in the way the current state of the process is packed and unpacked. In strong mobility the computational state is contained in the same package, in weak mobility the state (or part of it) is passed by parameters under control of the programmer.

In a classical migration scheme the application that migrates from host to host is temporarily halted and is restarted at the receiving host after the code is completely loaded and restored in its original form. See Table 1: Typical Migration Steps (page 26).

During steps 2-8 the application is not available for users or other processes that need to interact with it. After it is halted, it will become available again only when the migration process has completed. In time-critical applications, this may not be acceptable. In a control engineering environment, by example, the maximum time between the intakes of samples of the quantity under control is strictly defined and if the sample timing exceeds this threshold just one time this may compromise the complete control process.

Progressive mobility using component streams goes beyond the standard way of moving code by moving the application piece by piece from sender to receiver. During the migration the application continues to run and will be available to react to any event that will trigger an action. If the sequence and load distribution of the different executable components is well

chosen the migration can happen in parallel with its evaluation thereby almost completely eliminating network latency. It is a form of migration that even goes beyond *strong mobility* since the evaluation of the application will never be halted.

Experiments

We will choose examples to investigate the possibility to:

- Hide network latency
- Reduce invocation latency
- Reduce user interface latency
- Enhance application availability
- Reduce system latency by introducing parallel component evaluation

We expect that **network latency** can be hidden by moving each component during its idle time. Moreover we can try to start up the application before or during the actual migration. Since at arrival of the first component at the receiving host the evaluation of this component might start immediately also **invocation latency** should be reduced. If one of the first components is able to draw the user interface this will also reduce **user interface latency**.

By starting up the application at the sending host at the same time or before its migration is triggered we expect the **availability** of the application to increase.

In addition, since parallel component evaluation becomes possible during the migration phase, a reduction in **system latency** is expected for those applications that can take advantage of parallel evaluation.

In order to demonstrate the feasibility of progressive mobility using component streams we will first describe an experiment to **hide network latency**. We will provide some small existential examples in the programming environments Borg and Java.

Then we describe an experiment to **reduce system latency in low data rate environments**. This experiment will also be conducted in the Java environment.

Finally a similar experiment was set up but now focused on the **reduction of system latency by parallel evaluation.** This experiment was conducted in the Smalltalk environment to investigate the constraints of that environment too.

We will provide some small existential examples in different programming environments in order to show that for these applications the technique is useful. The determination of the universal nature of the techniques or the demarcation of the domain in which the technique proves useful is left for future work.

4.3 Proposed Technique

4.3.1 Basic Observations, Assumptions and Restrictions

As a first important observation we remind that the transmission over a network is inherently slower than compilation and evaluation and this will remain the case for many years to come (section 2.2.4.)

We remind also at the second observation, in the same chapter, that actual and future computer architectures provide separate processors for input/output (code loading) and main program evaluation (section 2.3.3).

A third observation is that many new applications are built following the principle of separation of concerns (e.g. object-oriented, components-based or aspect-oriented software development techniques). This leads to a modular design with relatively independent components. The applied paradigm will influence the granularity of these components. During the evaluation of the application processor control is passed from one component to the other while all the other components are idle.

We assume that the know-how and know-when of the migration of partitioned code is located in the sending host, so we apply a push strategy. However, this does not exclude the possibility of successful combinations with a pull-strategy but we did not implement this in our experiments.

The size of the applications in our experiments, especially the first one in Borg, are so small that they fall outside the window (see section 2.1.6 Window of Opportunity - page 39) in which the transportation time to send a block of code is directly proportional with the size of the block of code. This makes it impossible to reduce invocation and user interface latency at the receiving host since the transportation time to send the complete block of code is the same as the transportation time to send part of it.

However, from the perspective of hiding network latency and application availability the transportation time is not relevant. If we manage to migrate a component during its idle time, its transportation time is not important. From the perspective of the user or cooperating processes, the application remains available and is not influenced by transportation time constraints in the network.

Moreover, we will be able to reduce invocation and user interface latency at the sending host by keeping the application alive during its migration.

4.3.2 Technique Description

The introduced term *component streams* is inspired by streaming media but also by the transport mechanism for a sequential file, a stream, a data structure that allows only sequential access. During the streaming process the first part of the file will be already located at the receiving host while the other part of the file still remains on the sender platform. When streaming a running application, part of the application will already run on the receiving host while another part is still running on the sending host.

The application components described in this chapter are based on a limited component model where an application is build from different simple components that communicate with each other by sending messages and that does not involve events or other special architectural constraints and should not be confused with models that are more sophisticated as the J2EE Java component model.

Figure 29 shows the components of an application during the streaming phase. The lines indicate the communications between the components. The figure shows that an application becomes temporarily distributed during the streaming phase.

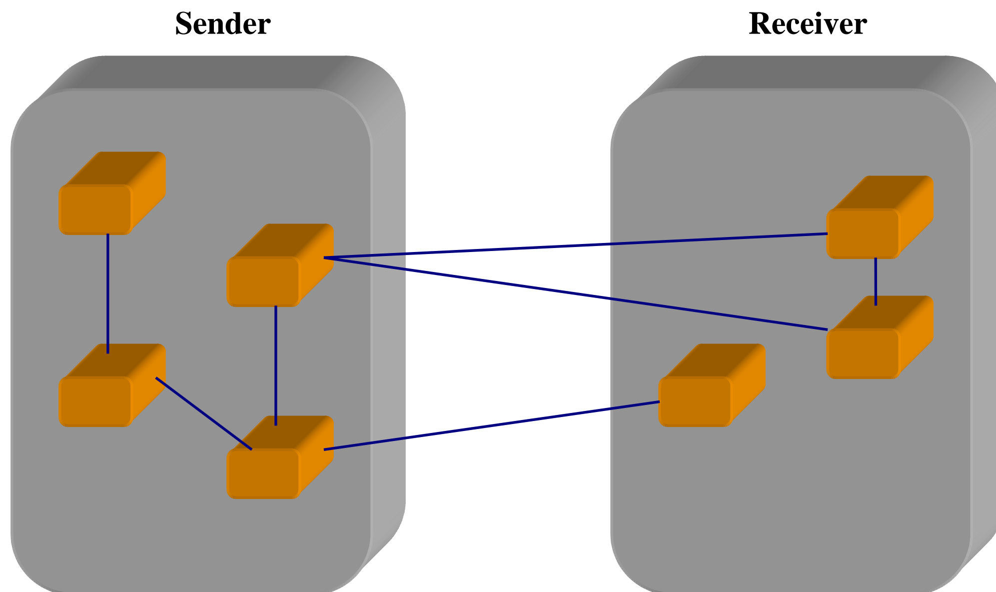

Figure 29: Components of an application during the streaming phase.

While the streaming unit for files is usually a byte or a word, for progressive mobility using component streams, the units need to be executable components and can take on a variety of forms: modules, functions, procedures, objects, agents, processes, threads and so on. If an executable component is sent over before the application has started it suffices to send over its code and start it up in the same manner as applets are loaded to a web browser and started (weak mobility). If, however, the application is already running before migration, one should send not only the bare code but also the intermediate values of the local variables of that evaluation unit and the information of the exact point in evaluation where the entity was stopped to be able to resume at the same point (strong mobility). This extra information is referred to as: *the computational state (including the runtime stack).*

The efficiency of the streaming process to hide network latency depends mainly on the *migration time* and *idle time* of each component. In the next sections we discuss how these component properties relate to each other.

4.3.2.1 Component Migration Time

The time a component needs to migrate from host to host is composed of the different times needed in the steps of Table 11.

$$T_{mig} \approx \sum_{i=1}^{9} T_i$$

Table 11: Migration Steps Time Intervals

Step(i)	Action	Time(T_i)
1	Halt the application	T_1
2	Pack it	T_2
3	Transform it	T_3
4	Transport to the receiver	T_4
5	Retransform it	T_5
6	Check it	T_6
7	Unpack it	T_7
8	Adapt it	T_8
9	Resume the application	T_9

The transport time T_4 depends mostly on the data rate B of the communication channel because this is mostly much lower than the clock speed of the sending or receiving host. The other times depend on the clock speeds C_{sender} and $C_{receiver}$ of the sending and receiving host processors, respectively (see units in Table 12).

Table 12: Deployed Units

base quantity	symbol	unit
Data rate	B	bps
Clock speed	C	Hz
Number of bits	b	bits
Number of instructions	I	instructions

If we call b_4 the number of bits transported and I_i the number of instructions needed in step i (Table 11) then the migration time T_{mig} becomes approximately:

$$T_{mig} \approx \frac{b_4}{B} + \frac{\sum_{i=1}^{3} I_i}{C_{sender}} + \frac{\sum_{i=5}^{9} I_i}{C_{receiver}}$$

If $C_{sender} = C_{receiver}$ and if we call

$I_{tot} \approx \sum_{i=1}^{3} T_i + \sum_{i=5}^{9} T_i$ then Figure 30 shows the migration time of a component of 1KiB[10] if 10^6 instructions are needed for halting, packing, transforming, retransforming, checking, unpacking, adapting and resuming the code.

[10] One KiB = 1024 Bytes

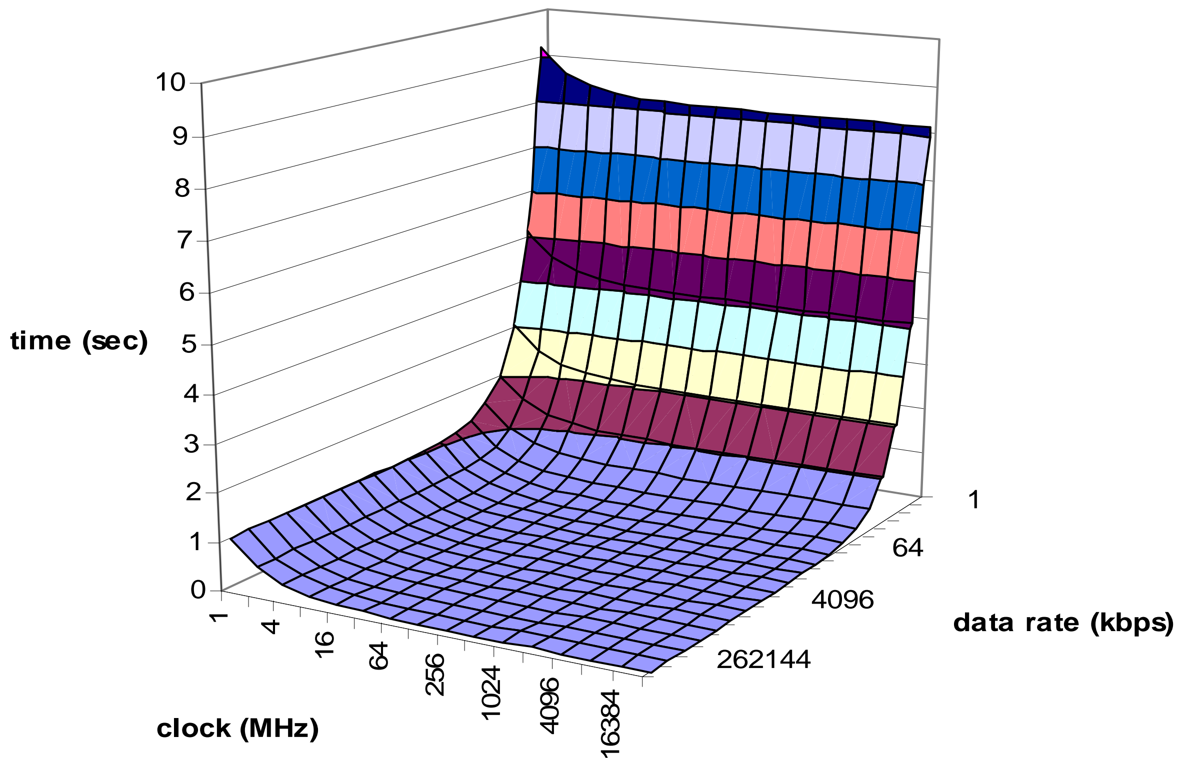

Figure 30: Migration time for 1KiB code and I_{tot} =10^6

The figure shows that the total migration time depends mainly on the transport time T_4 and thus the available data rate, for CPU clock frequencies up to 16 GHz[11].

4.3.2.2 Component Idle Time

During the evaluation of an application that is built from components, the task of the application will be performed by the different components. In many languages the component structure reflects a functional decomposition of the application. During the control flow of the application the work is done by different components mostly one at a time while all others remain idle. If we assume as a first and rough approximation that the workload of an application is equally divided over all its components and the application runs in a single thread, the time a component remains idle depends on the number of components and the evaluation clock speed.

Figure 31 shows the idle time per component in function of the system clock speed if we assume an idle time of 100 seconds at 1 MHz clock speed. The graph shows the relation between the idle time of a component and the speed of the components if the total workload is equally divided between all components. If your competitors work twice as fast, your idle time becomes half of the original one.

[11] Intel expects to deliver CPU's with a clock frequency of 24 GHz in 2007

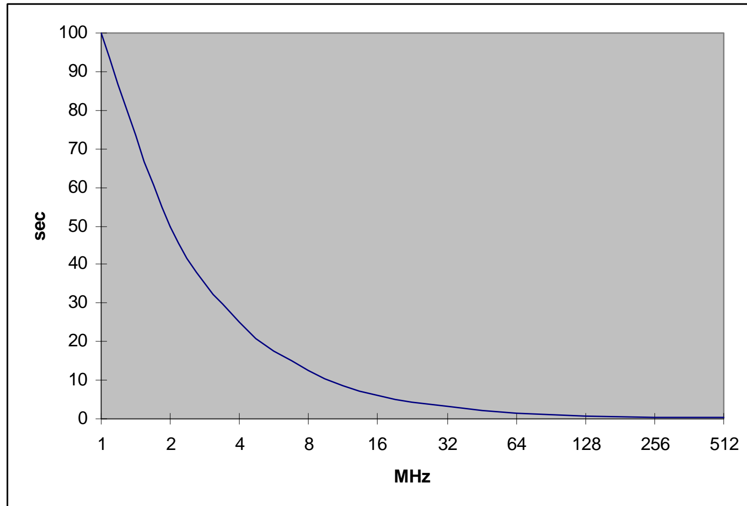

Figure 31: Component idle time versus evaluation clock speed

Figure 32 shows the idle time per component if the evaluation of the application takes 100 seconds. If the number of workers increases to *n* for the same amount of work each of the workers needs only to work $1/n^{th}$ of the original time. The remainder of the time becomes idle time. In practice the idle time will increase even faster since an increase of the number of components tends to make an application less efficient, and therefore more time-consuming due to the introduction of inter-component communication overhead.

If the workload is not equally distributed, as we may expect from real world applications the times should be interpreted as average times.

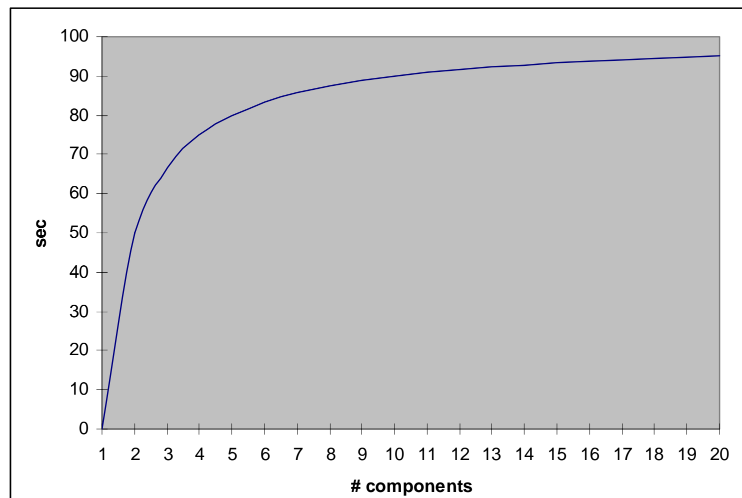

Figure 32: Component idle time for 100 sec application

4.3.2.3 Necessary Conditions for Removing Network Latency

In order to be able to move every component in parallel with the evaluation of the application the following conditions (Table 13) must be satisfied:

Table 13: Necessary Conditions for Removing Network Latency

1	Each component must have at least one period of idle time equal to or greater than the time the component needs to migrate.
2	The relative point of time where this idle time period starts must be known in advance
3	If different components have only one free slot of idle time equal to or greater than the time that component needs to migrate, these slots may not overlap

If all these conditions are satisfied it suffices to migrate the components at the point of time where their idle period starts.

If we build a new application that should be able to move by component streams, these design rules should be kept in mind. It will not always be possible to comply with them completely, but the more we approximate them the more the application will benefit from the proposed technique. If we need to stream an existing application, we may need to adapt it to comply better with the above conditions.

If the first condition is not met, the technique can still be deployed but migration of the application will then cause some delay in its evaluation. We expect however that in many cases architectural transformations could be applied to transform the original application to an equivalent one that complies better with the first condition.

If the second condition is not met the migration of the application will also cause some delay in its evaluation. If the exact onset of the idle time is not known in advance it will be possible in some cases to estimate the delay based on statistics obtained from application profiling. Modifying the application at its design level could transform the original application to an equivalent one that complies better with the second condition.

If the third condition is not met the migration can only be optimized for one of the conflicting components although here also architectural transformations at the design level may resolve the conflict.

4.3.3 Migration Strategies

Progressive mobility using component streams will move a mobile application piece by piece from sender to receiver. During the migration the application continues to run and will be available to react to any event that will trigger an action. It is important that the sequence of the different components is guided in such a way that the migration can happen in parallel with its evaluation thereby eliminating the network latency almost completely. We describe some typical strategies below.

4.3.3.1 Self Triggered after Last Instruction

The first strategy lets each component trigger its own migration just after it releases its control to another component (Figure 33). This is a simple strategy that can be deployed if the workload of an application is more or less equally divided over its components and if the number of components is sufficiently large so that the average idle time is high and average migration time is low. The strategy implies that the underlying framework is powerful enough to allow the components to migrate completely autonomously. One environment that provides such autonomous components is the mobile multi-agent environment developed at our lab: Borg [Van Belle et al. 2001].

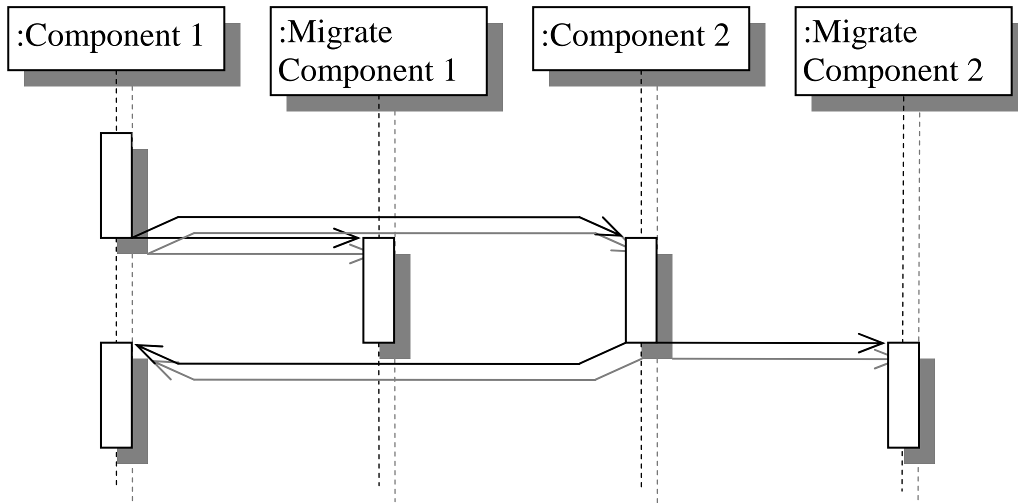

Figure 33: Migrate after last instruction

4.3.3.2 Self Triggered based on Profiling

The second strategy assumes the existence of a profiling process (Figure 34). The profiler is an independent process that runs in parallel with the application built from the different components. During the evaluation of the application the profiler generates a statistical profile of the application behavior. The appearance of the profile could be a dictionary containing the different evaluation contexts of a component as a key and the average idle time following the evaluation in this context as value. Each component will at the end of its evaluation consult the profiler to find out if the current moment in time is appropriate to migrate. The main disadvantage of self triggering is the extra time the components need to spend after their evaluation.

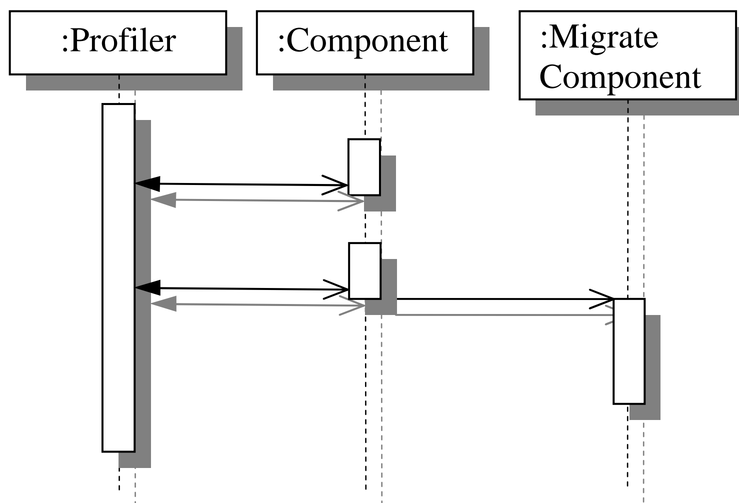

Figure 34: Self triggered migration based on profiling

4.3.3.3 Under Control of a Supervisor

If a profiler is running in parallel with the application it is advantageous to transfer the migration control to this process which in this case we like to call a *supervisor* (Figure 35). The components itself are now freed from checking the opportunity to migrate each time they

run. It is the supervisor that will decide when to migrate based on the profile gathered so far, statistical data from the past, current host performance differences etc…

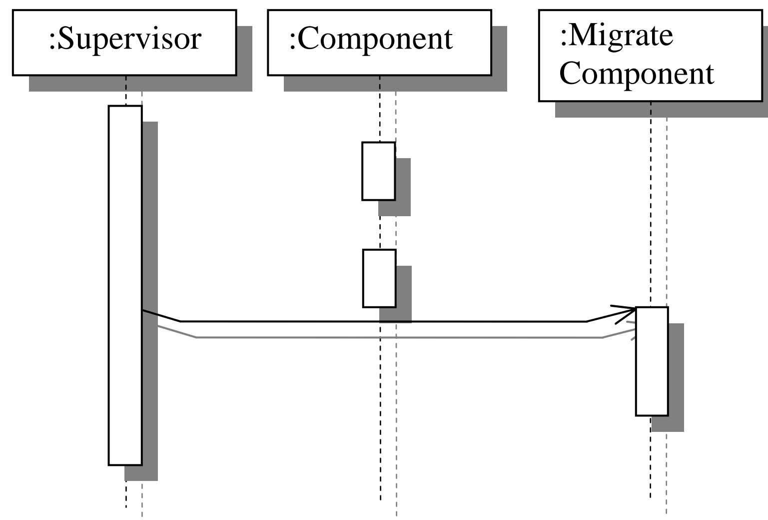

Figure 35: Migrate under control of a supervisor

4.3.3.4 Fixed Migration Strategy

If new applications are developed from scratch, the developer can model the design to optimize its potential for progressive mobility using component streams. The development environment can provide support for that. The developer can use his knowledge of the high level purpose of the application to describe a fixed migration strategy of its components including the exact moments in time where a migration should start. If the application decides to migrate, or if another component asks the application to do so, a supervisor component, running in parallel with and independent from the application, will guide the migration of the application. The supervisor will trigger the migration based on fixed rules set up by the developer. If the application needs to migrate more than once during its lifetime the supervisor has to migrate with the application.

4.3.3.5 Dynamic Migration Strategy

If there is no fixed strategy available, the supervisor component, running in parallel with the application can do the profiling of the application's behavior in the same sense as described in section 4.3.3.2. If the application needs to migrate, then the supervisor will guide the migration of the application based on the profile obtained so far.

4.3.4 Discussion

The simplest strategy: *self triggered after last instruction* (section 4.3.3.1) can only be applied if the underlying platform provides strong migration and complete autonomy to its components. Moreover, the accompanying naming and routing system should be able to maintain transparently the connections between the components. These are properties that can only be found in some mobile multi-agent platforms but not in more current program environments as Java. The second strategy: *self triggered based on profiling* (section 4.3.3.2) needs the same facilities since here too, a component is supposed to migrate itself autonomously after it consulted the profiler.

Therefore the only strategy that we can apply in a classic programming environment is: *under control of a supervisor* (section 4.3.3.3). This strategy is potentially the most efficient of all since this strategy does not introduce extra code in the components themselves but can run in parallel with the application, eventually on a separate processor.

4.4 Experiment to Hide Network Latency

4.4.1 Borg Environment

Given the complexity of the implementation of this technique, we first start by building a proof of concept in Borg, an environment that by its nature facilitates the expression of mobile components. The Borg environment allows that components migrate completely autonomously which give us the opportunity to apply the self triggered after last instruction strategy.

The Borg environment models applications as cooperating autonomous agents as shown in Figure 36.

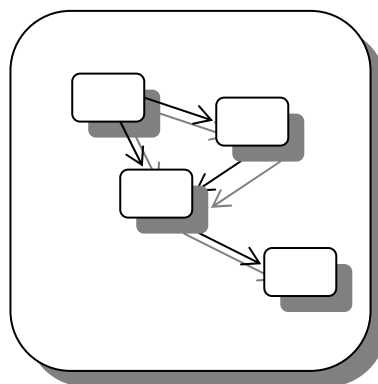

Figure 36: Cooperating agents

This appears to be a natural environment to establish an object streaming proof of concept since it allows us to migrate the application components (the agents) one by one. Figure 37 shows a snapshot after two of the components from the application in Figure 36 have been migrated. The migration can be triggered by the component itself or under control of another component.

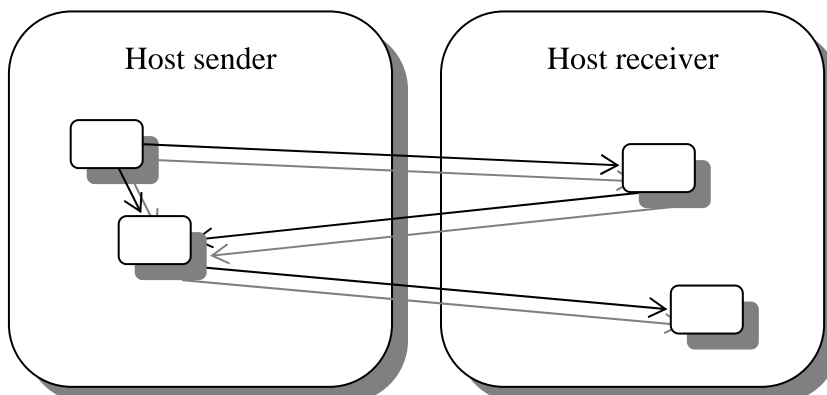

Figure 37: Components during the Migration Process

An ideal migration strategy would be obtained if each component could be moved during its idle time by a separate processor. This can be done by making sure that the I/O processor runs parallel with the main processor or by providing each component of the (now distributed) application with a separate host (Figure 38). If every component migrates during its own idle time without claiming processing power of the application itself, the application can stream from one set of sending hosts to a set of receiving hosts without the burden of network latency.

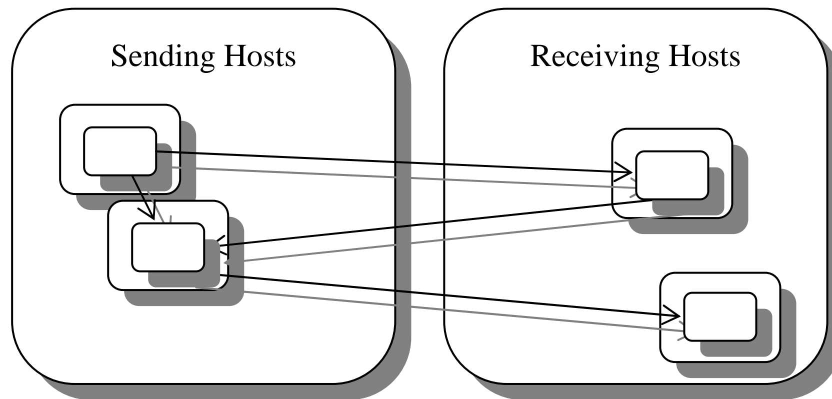

Figure 38: Components during the Streaming Process

As a proof of concept we implemented a simple Borg application (Figure 39) that moves two components C1 and C2 (Figure 40) from their sending hosts to two receiving hosts. The only task of each component is counting to 20000 and then signaling a clock agent that it has finished its job and passing control to the other component which in turn will go trough the same procedure. The count of 20000 was chosen to make sure that the idle time of each component is greater than its migration time. The separate clock agent is introduced to be able to log timing events on different components in a distributed environment. The introduction of a separate timing agent avoids severe synchronization problems between the distributed components. Each component will start to migrate after it finished its counting job and during this time the other component does the counting.

4.4.1.1 Implementation

Figure 39 shows the Borg code. Basically it creates an agent **a1** that can move from host **a** to host **c** and vice versa and an agent **a2** that can move from host **b** to **d** and vice versa. Then each agent is told to remember that the next agent to do the counting is the other one and agent **a1** gets the message to start the *work*.

The *work* method of each agent starts a counter from 0 to 19999 and then the other agent is set to work, a time log is send to the **clock** agent and the working agent migrates to the other host.

```
{
show_clock(ref): display(ref, " ", clock(), eoln);
clock:agent("hostclock");

a:agent("hosta");
b:agent("hostb");
c:agent("hostc");
d:agent("hostd");

create_agent(num, places) ::
{ next_agent :0;
  pcount:1;
  set_next(a): next_agent:=a;
  work(): { for(i:0, i<20000, i:=i+1, void);
            next_agent->work();
            clock->show_clock("loop done"+text(num));
            pcount:=(pcount\\size(places))+1;
            agentmove(places[pcount]);
            clock->show_clock("move done"+text(num))
          };
  agent:clone2agent("component"+text(num));
  agent->agentmove(places[pcount]);
  agent
};

a1:create_agent(1, [a,c] );
a2:create_agent(2, [b,d] );
a1->set_next(a2);
a2->set_next(a1);
a1->work();
}
```

Figure 39: Borg proof of concept code

Figure 40 illustrates the hierarchical naming/routing structure which is chosen in such a way that the path between the sending and receiving hosts is of equal length i.e. trough the Timing Host and that the path from the components to the timing host is as short as possible (i.e. directly to the Timing Host). In order to provide each component its own processor we used five different hosts[12].

[12] Each host comprises a Gentoo Linux environment running on a 1800 MHz AMD processor with 256 MB RAM.

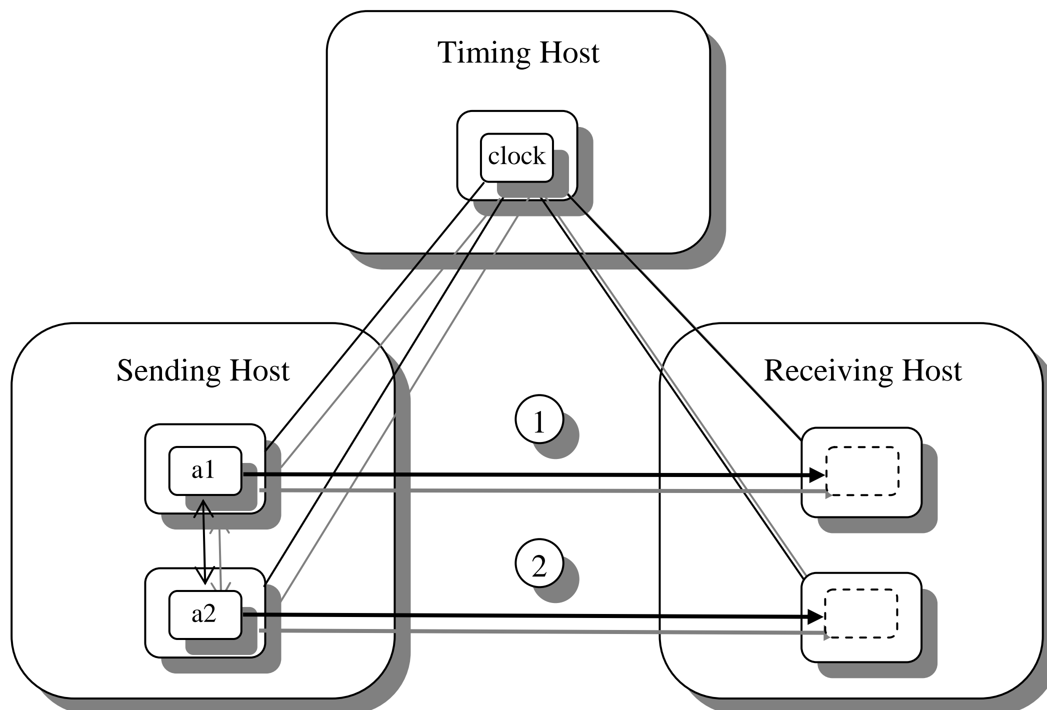

Figure 40: proof of concept setup

4.4.1.2 Results

We allowed the two components to work 500 times without migration and then again 500 times with migration and calculated the average time the application needed to complete. We calculated the average time in order to flatten out unpredictable time variations introduced by the Borg garbage collection, network data rate variations and/or other possible unpredictable events.

The average time the application needed to complete without migration is **0.153 ms** with a standard deviation of 0.076 ms.

The average time the application needed to complete with migration is **0.106 ms** with a standard deviation of 0.047 ms.

4.4.1.3 Discussion

Apparently the application runs even faster if it migrates at the same time. Borg agents use a polling mechanism to check for new messages in their incoming message queue. During the migration of an agent this polling mechanism is suspended, this may explain the gain in time if the application is migrated.

In either case the experiment showed that it is possible to migrate this specific running application without slowing it down, as if there where no network latency at all. Even better, the migrating application runs faster than the without migration. Although we do realize that in real-world, non-distributed applications we might expect the application to slow down somewhat during the migration.

4.4.2 Java Environment

Encouraged by the proof of concept results we implemented the use of component streams in Java, a more established environment. We report on two more extended experiments in this

environment [Devalez 2003]. Java components are not able to migrate autonomously, therefore we choose to implement a fixed migration strategy under control of a supervisor, which appears to be the optimal architecture and migration strategy for the chosen experiment.

As an example of the fixed migration strategy under control of a supervisor we migrate an fractal generation application JULIA 2 [Devaney 1992] that plots a fractal on a screen. The application is designed to consist of two largely independent components, one responsible for the calculation and one for the graphical presentation of the calculated points to the screen.

Our main concern here is the efficiency of the migrating process itself, therefore we designed the fractal application in such a way that it can stream efficiently to the receiving host. The application is composed of two components, a component responsible for calculating the fractal *(calculateFractal)* and a second component responsible for coordinates conversion, scaling, plotting etc. *(plotFractal)*. The total workload of the application is, as much as possibly, equally shared by the two components.

To minimize the invocation latency [Krintz et al. 1998, Stoops et al. 2002] we send the graphic presentation component first so that a human observer at the receiving hosts will have a quick visual response after the migration starts. On the other hand, if we assume that the receiving host has much more computing power, it could make more sense to migrate the CalculateThread first to reduce the total evaluation time.

4.4.2.1 Implementation

Both components are implemented as a movable thread (*PlotFractalThread* and *CalculatingThread*), running on a µServer (Figure 41). Threads run in their own namespace and there is no standard mechanism that allows them to communicate with each other, so we introduce an RMI object *SharedQueue* to allow the two threads to pass and retrieve information from. *CalculatingThread* puts its results on a two-dimensional array of 50 pixel coordinates on the queue datastructure while *PlotFractalThread* polls the queue to get its input points.

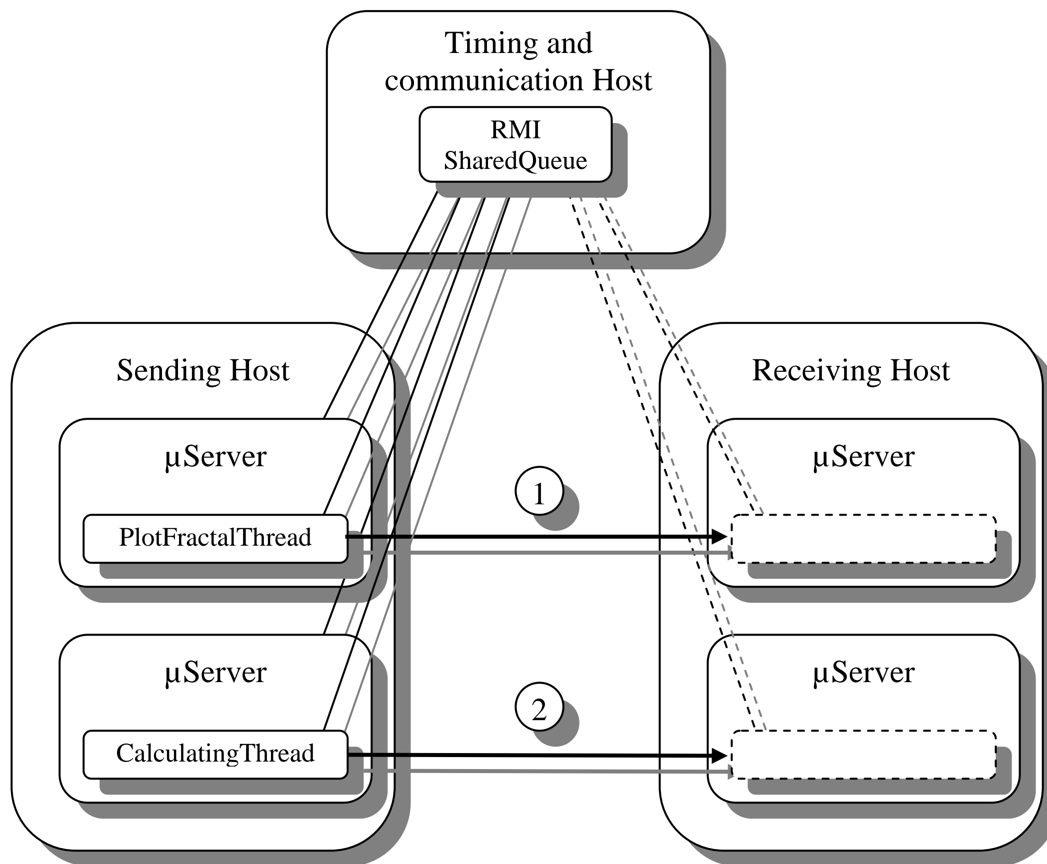

Figure 41: Java experiment architecture

The µServers play the role of (distributed) supervisor and since the two components run on top of their µServers it is not possible to run the supervisor independently on a separate processor in this setup. In this example we apply a fixed migration strategy, the sequence and time of migration is hard-coded in the supervisor. The supervisor interrogates on a regular basis the status of the *SharedQueue* object and decides when the components are moved. In our setup *PlotFractalThread* is moved immediately while the *CalculatingThread* starts its calculations. If the supervisor detects that there are 300 elements available in the *SharedQueue* it migrates the calculating thread. Since migration happens only once there is no need to migrate the supervisor as well.

In order to provide each component its own processor so as to obtain true parallelism between the migration of one component and the evaluation of the other at the sending and receiving side we used five different hosts[13], four µServers to host the two components and one central RMI host for time logging and facilitating thread communication (Figure 41).

4.4.2.2 Results

First the application was evaluated without migration. The application was launched 30 times and we calculated the average time in order to flatten out unpredictable time variations introduced by the garbage collection, network data rate variations or other possible unpredictable events. The average time to complete the application without migration was **2 sec 566 ms** with a standard deviation of 20 ms.

[13] Each host comprises a Gentoo Linux environment running on an 1800 MHz AMD processor with 256 MB RAM. The hosts are interconnected via a 100 Mbps LAN network.

Then we launched the application again 30 times but now the application was migrated from the sending host to the receiving host immediately after it was launched. The average time to complete the application now with migration was **2 sec 530 ms** also with a standard deviation of 20 ms.

The experiment showed that it is possible for a supervising component to migrate a running application without slowing it down, as if there where no network latency at all. Even better, the migrating application even runs slightly faster than the same application without migration.

The µServer at the sending host maintains a logging process to find out when certain events take place. This logging process runs only at the sending host and is not necessary at the receiving host. This explains why in this setup the application runs faster if it is migrated at the same time since the parts at the receiving host are not hindered any more by this polling process.

4.4.2.3 Discussion

Threads

To divide our program in different processes, we used the available Thread class in combination with the µCode toolkit to move the threads to the receiving host. However, threads cannot communicate with each other directly, because they run in different namespaces. We introduced the SharedQueue object to work around this problem. This slows down the original application, because all communication has to go through this object. On the other hand, this approach eliminates the direct connection between the migrating components, which makes it easier to transfer them. In order to minimize extra slowdown from the SharedQueue object, we ran it on a separate processor.

RMI

Since we migrate the threads to an other host, the SharedQueue object must also be available for the threads running on the receiving host. Therefore we decided to make SharedQueue accessible through RMI. The disadvantage of this is that a local stub must be available on all the hosts that use the remote interface SharedQueue.

Reflection is very useful for its intercession part. In the context of mobile code, it would allow us to change references to objects into remote references once the objects are transferred. We would be able to introduce meta-objects that observe the running application and change the object references when needed. Java already supports a limited introspection part of reflection in the java.lang.reflect library. Unfortunately Java does not support intercession. In our experiments, we worked around this problem using RMI. Since the SharedQueue object does not move, we did not have to change references to this object. The threads we migrated also did not have direct references to one another, which makes the use of reflection in this case unnecessary. But when applying our techniques to other applications, we will have to consider reflection to change references at the meta-level. Reflex [Tanter et al. 2003] is a valid candidate to introduce intercession in the Java environment.

Frames

To draw our fractal figure, we used the java.awt.Frame class. We can add an object which extends the Canvas class and overrides the Canvas paint() method to draw pixels on it. When a Frame is shown on the screen, it immediately invokes the paint() method from the object in the frame. And every time the Frame has to be redrawn, the paint() method is invoked again.

In our experiment, we added the PlotFractal thread to the Frame, since it overrides the paint() method to draw the pixels. This made it difficult to synchronize the threads, because the PlotFractal thread has to show the Frame first, and then wait some time until the Calculating-thread finishes calculating the pixels, and then draw the pixels. With every redraw of the Frame, this starts all over again. A possible solution to this is to keep the content of the Frame, so that whenever we restart the thread at arrival, it keeps the pixels that are already drawn. This would improve our experiment but re-initializing the thread also creates a new Frame, and the graphics needs to be redrawn again anyhow.

µServers

To send a thread to a receiver, we need to activate a µServer that runs on the same processor as the thread. We cannot make them run separately because we need to pack and unpack the thread we want to send.

If we use the µServers as supervisors, they will use some of the processor time, which slows down the application. After the migration, the application will run faster, since the receiving µServers do not evaluate the extra supervisor instructions. This is a drawback of using threads as migrating objects.

If we would have used autonomous agents, which contain everything needed to transfer them, we would not see a difference in time between an application running on the sending hosts and an application running on the receiving hosts. This also explains why the time to complete the migrating application is less than when the application is not migrated.

4.5 Experiment to Reduce System Latency in Low Data Rate Environments

The previous experiment showed that component streaming does not slow down the application, but it does not show the behavior of an application when it is migrated in low data rate environments. Therefore we did setup a data rate simulator and added a graphical user interface to the application to study its behavior.

4.5.1 Implementation

To be able to run experiments on selected data rates we need to put some simulation system in place. In this section we describe how we implemented different data rates in the Java environment.

4.5.1.1 Finding the Size of Objects

One way to simulate data rates is by blocking the migration a certain amount of time, depending on the size of the objects we want to migrate. To do this we will have to create our own outputstream, so that every time an object is serialized, the size of the object is calculated and the migration is delayed for the time needed to transfer it. In Smalltalk, there is a sizeof-method available to get the size of objects directly. In Java, a fast way to get the size is by serializing the objects to a ByteArrayOutputStream first. Then we can get a byteArray from

this stream, and ask the size of it. This will give us the size of the serialized object. As we have to serialize the objects to an ObjectOutputStream afterwards, this means we have to serialize them two times, as the ByteArrayOutputStream does not keep the object information. But since serialization is an available feature of the Java language, it is the most efficient approach for determining the size of objects, so we will use this in our experiment.

4.5.1.2 A Data rate Simulating OutputStream

We created our own outputstream, we will call it the DRLOutputStream (Data Rate Limiting Output Stream). This DRLOutputStream inherits from the ObjectOutputStream, and overrides its methods to wait some time before serializing them. The time to wait depends on the size of the objects, and on the data rate, which is passed when starting our application. To suspend the migration, we must suspend the process responsible for migrating the components. This can be done using the sleep method from the Thread class. After waiting some time, we will then really serialize the objects by invoking the ObjectOutputStream methods. We replaced the ObjectOutputStream object used in µcode by our DRLOutput Stream. Different data rates can now be simulated by specifying them when starting our application. This will be useful to test the benefits of progressive mobility using component streams.

4.5.2 Adding a Graphical User Interface

In order to show the power of progressive mobility using component streams on low data rates, we decided to extend the fractal drawing application with a user interface. This interface consists of Swing components (labels, buttons and text fields), and a JavaCanvas for displaying the fractal. The user will be able to change the parameters for the fractal, and to zoom in on certain parts of it. The area that will be shown while zooming in can be chosen by moving a rectangle over the drawn fractal. We took a screenshot of this application showing the graphical user interface and several command line windows to monitor the threads (Figure 42).

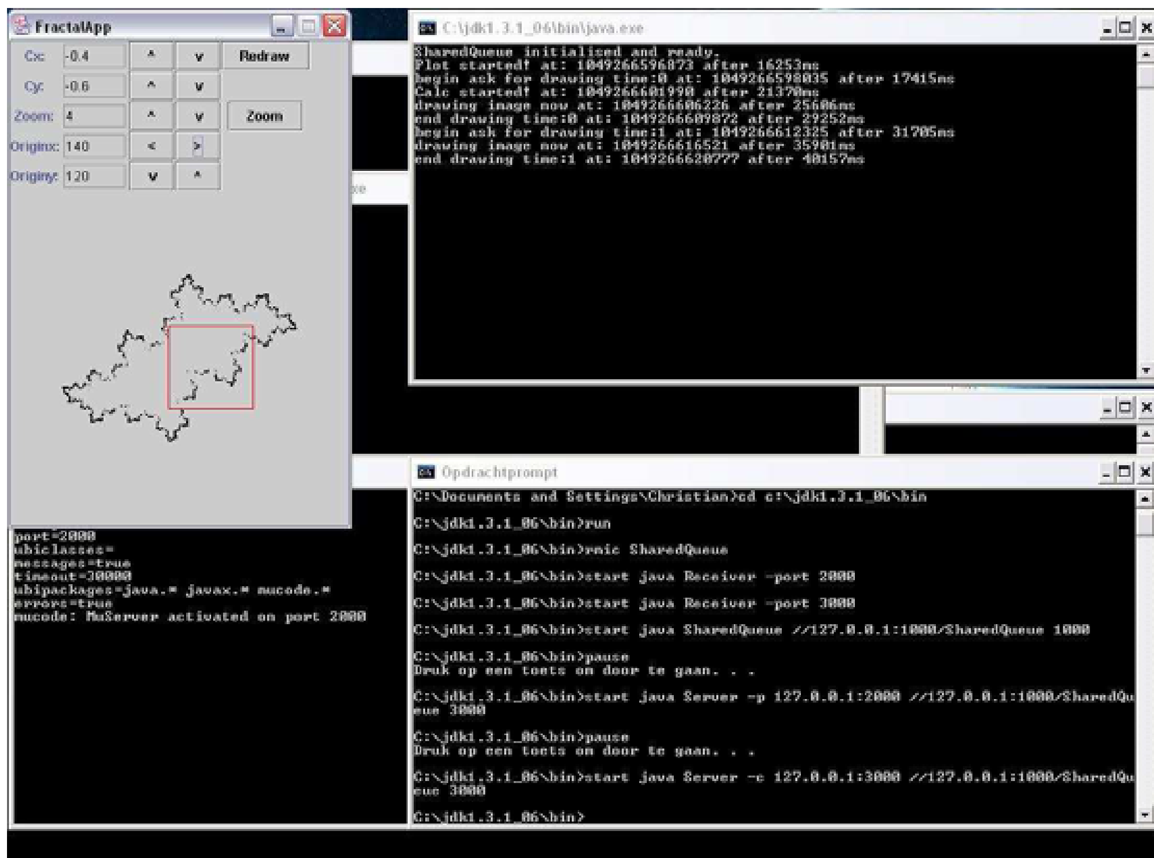

Figure 42: Graphical user interface and thread windows

Applying a user interface will show the availability of the application while transferring the other components that are part of the same application. The user will be able to redraw, zoom in, or change the different parameters of the fractal while the components are migrated.

4.5.3 Strategies

We implemented different migration strategies in our experiments. We used a fixed migration strategy using a supervisor, and a strategy involving a profiler.

In order to be able to compare application component streams and normal code migration we also migrated the components of the application without starting them on the sending host first. This strategy implements the normal download of applications, where only source code is transferred, and then started at the receiver.

Some initial experiments to test our code however showed that we must be careful when transferring an initialized user interface object, because this object can suddenly become a lot larger than a non-initialized one, and can take much longer to migrate. Since we want to compare the migration of running components and components that are not started yet, we needed our application to send objects of almost equal size and with almost equal migration time. Therefore we implemented our application to do just this by applying the technique of progressive anticipative mobility using proactive migration, a technique that we will further explain in chapter 5 of this dissertation.

Basically the application now migrates a non-initialized copy of the running component and after the migration the computational state is adapted to reflect the changes that occurred at the running version on the sender. In this set up this was not too difficult since most of the computational state is stored in the distributed SharedQueue object.

Again we provide each component with its own processor to allow the migration of one component to run in parallel with the evaluation of the other component[14].

The setup is shown in Figure 43 which is similar with Figure 41. In this setup however the PlotFractalThread will also contain the settings of the user interface, and will react on user interactions. The SharedQueue will also be extended to contain extra logging information to be able to adapt the computational state of the threads at arrival to reflect the changes that occurred at the running version at the sender during initialization of the user interface object.

We simulated three low data rate environments: a 2400 bps data rate, as still used in some maritime networks, a 9600 bps data rate, as in cellular phone modem networks, and a 56 kbps data rate, as in the standard modems that are still used for dialup lines. We performed experiments with three strategies:

- **normal**: Just send the components and start them up at the receiver
- **profiler**: Thread decides to migrate based on number of points already put in queue
- **supervisor**: Threads are migrated by a supervisor that polls the queue on a regular basis to check when number of points in queue reach a certain minimum

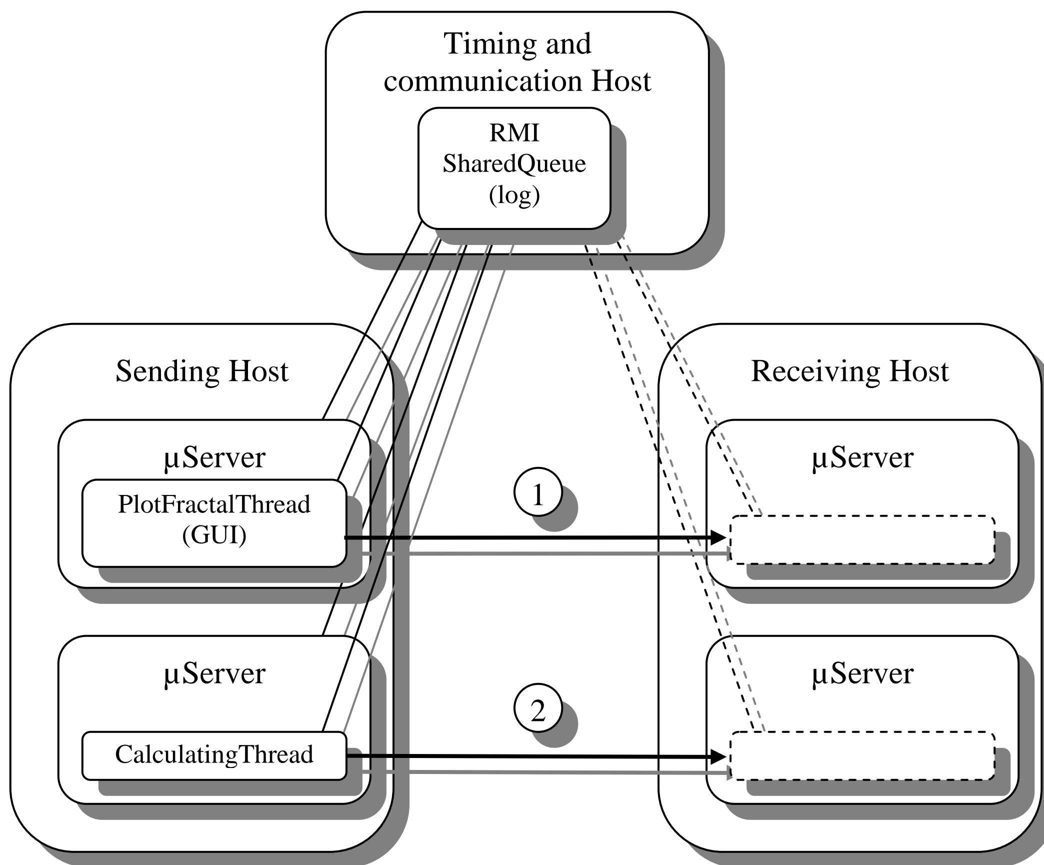

Figure 43: Java experiment with GUI architecture

Figure 44 shows the sequence diagram of the experiments.

---

[14] For each host we used a Gentoo Linux environment, running on an 1800 MHz AMD processor with 256 MB RAM.

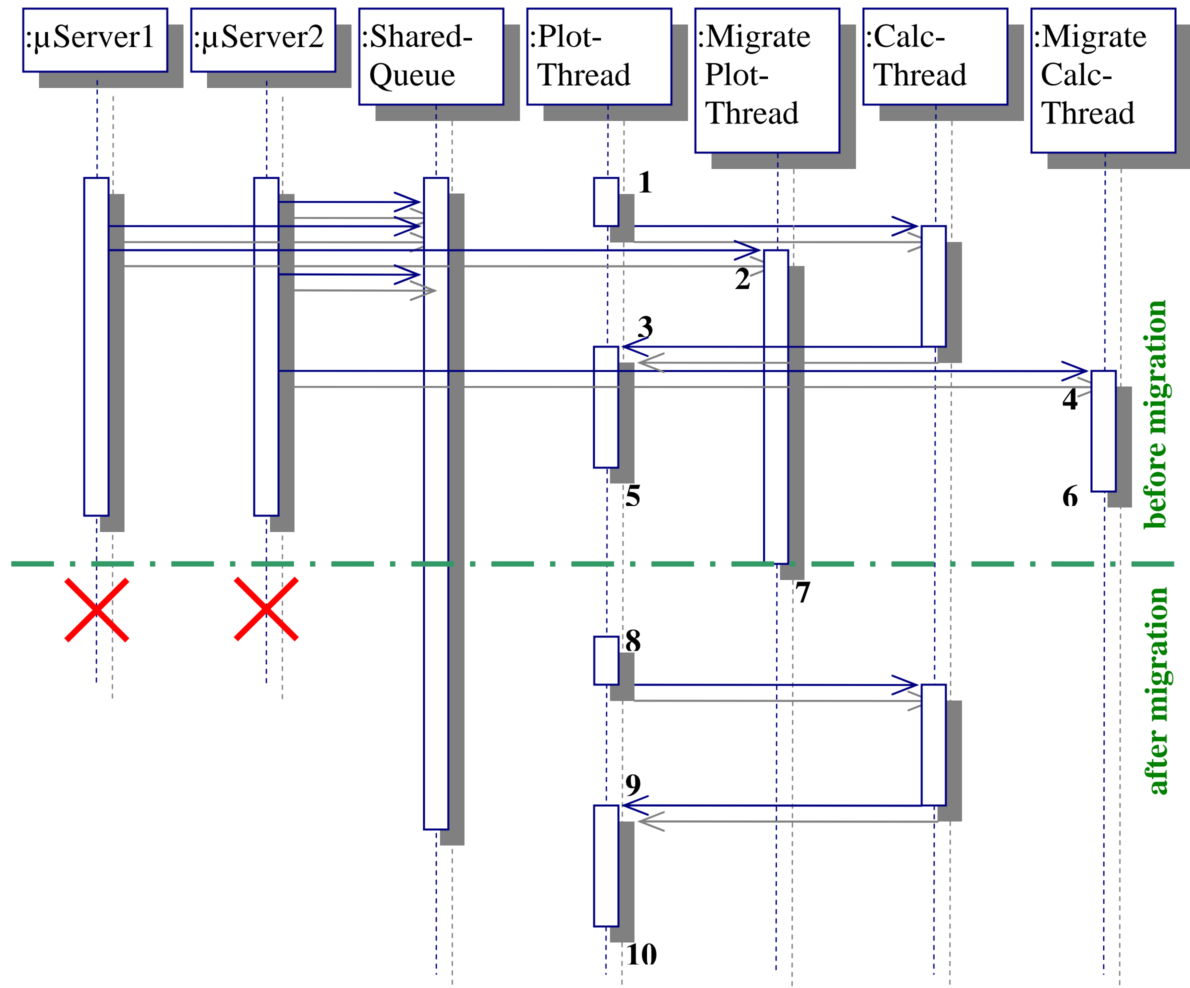

Figure 44: Sequence diagram of the extended fractal draw experiment

The numbers on the sequence diagram indicate the moments on which we logged the time in the SharedQueue.

In the sequence-diagram can be seen that the µServers at the senders run in parallel with the application and, check the SharedQueue at regular time intervals to get the state of the components. They also can act as supervisors and trigger the migration of the components, and after these components have migrated to the receivers, they halt their evaluation. The SharedQueue always stays available, since it logs the time and is needed for the communication between the threads, even after their migration. We will now describe the sequence-diagram in more detail at the different moments. The numbers correspond with the numbers found in Figure 44.

1. The PlotThread asks to draw the fractal, because it received a redraw event by the user, or because the application started. It then notifies the CalculateThread that it can start calculating new points (using the SharedQueue). The PlotThread waits until the CalculateThread finishes.

If the profiling strategy is used then CalculateThread halts when it has calculated enough new points, if the supervisor strategy is used it is the profiler that checks on a regular base the number of points calculated and that will halt the CalculateThread when the number of new points reach an certain minimum (300 in this setup).

2. While the CalculateThread is busy, the PlotThread starts migrating to the receiving host. This can take more time than the time needed to calculate the points, as the PlotThread contains the user interface, and is therefore a bigger object to migrate.

3. The CalculateThread finished calculating the new points and notifies the PlotThread that it can start drawing. The migration of the PlotThread is not finished yet so it will be the original version that is still available on the sender that will draw the fractal graph, while a copy of this thread is migrating to the receiving host.

4. While the PlotThread is busy drawing, the CalculateThread is also migrated to the receiving host. The migration of this component is usually shorter than the PlotThread, since it is a smaller object. As shown in Figure 44. This migration happens in parallel with the migration of the PlotThread and the evaluation of the original PlotThread at the server.

5. The fractal was drawn on the sending host platform. The components are now idle and wait for the next redraw.

6. The CalculateThread was sent. From this moment on, the µServer on which it was running halts, and the component starts evaluating at the receiver.

7. The PlotThread was sent, and starts evaluating at the receiver now. The µServer on which it was running halts since the services as possible supervisor or migrator are not necessary anymore.

8. A new redraw event makes the PlotThread ask for a new drawing. This time, the redraw takes place on the receiving host platform. The PlotThread notifies the CalculateThread to start.

9. The CalculateThread finished calculating and notifies the PlotThread to draw.

10. A fractal was drawn on the receiving hosts.

The sequence-diagram is similar for the different strategies we used, except for the normal sending strategy where the components are not started at the sending host. It shows us that the user might have some extra time during migration on which he can still use the application. We will confirm this in the results of our experiments. To eliminate time variations due to external influences, we calculated the average over several runs of the same experiment.

4.5.4 Results

We applied the different strategies and data rates on our experiments. We will now compare the results from these experiments.

4.5.4.1 Time Needed to Finish the Application

In chapter 3 we noted that applications can finish earlier if parts of the application are sent progressively to a receiver and if we manage to startup the application at the receiver before the complete application is migrated. In applying progressive mobility using component streams, we also may evaluate a part (a component) of the application on the receiver before the complete application is migrated, so here too an application can finish earlier. Even if we do not start the application earlier at the receiver we will discover that in this experiment the application ends earlier if it is migrated.

As we did in our previous experiments, we logged how much time was needed to finish our application while evaluating it with and without migration. We also logged the time while evaluating on the receiver. Table 14 shows the average time for each strategy; Table 15 gives an indication of the standard deviation of the times measured. As can be seen in Figure 45, the

supervisor seem to influence the evaluation time of the application on the sender, since it takes more time to complete the application in comparison to running it on the receiver. This was something we could expect, since we were using the µServers that are running in parallel with the components on the same processors, as supervisors. The results however show that while migrating the application, it does not slow down. In fact, it seems to run even faster. This is probably because when the application is migrating the supervisor is halted because it does not need to check on a regular base the number of points anymore. The extra clock cycles previous needed by the supervisor can now more efficiently be used by the application. We can conclude from these results that in this experiment the application was not slowed down during migration, as we already experienced in our previous experiments. Another observation is that the application seems to run somewhat faster when using a profiler in comparison to running it using a supervisor. If a profiler strategy is applied the CalculateThread can count the number of generated points by itself and doesn't need the profiler to monitor the shared queue.

Table 14: Average timing results (ms) to finish the application

strategy	on sender	while sending	on receiver
profiler	4168	3490	3288
supervisor	4676	3831	4320

Table 15: Standard deviation (ms) of timing results to finish the application

strategy	on sender	while sending	on receiver
profiler	240	609	544
supervisor	227	482	945

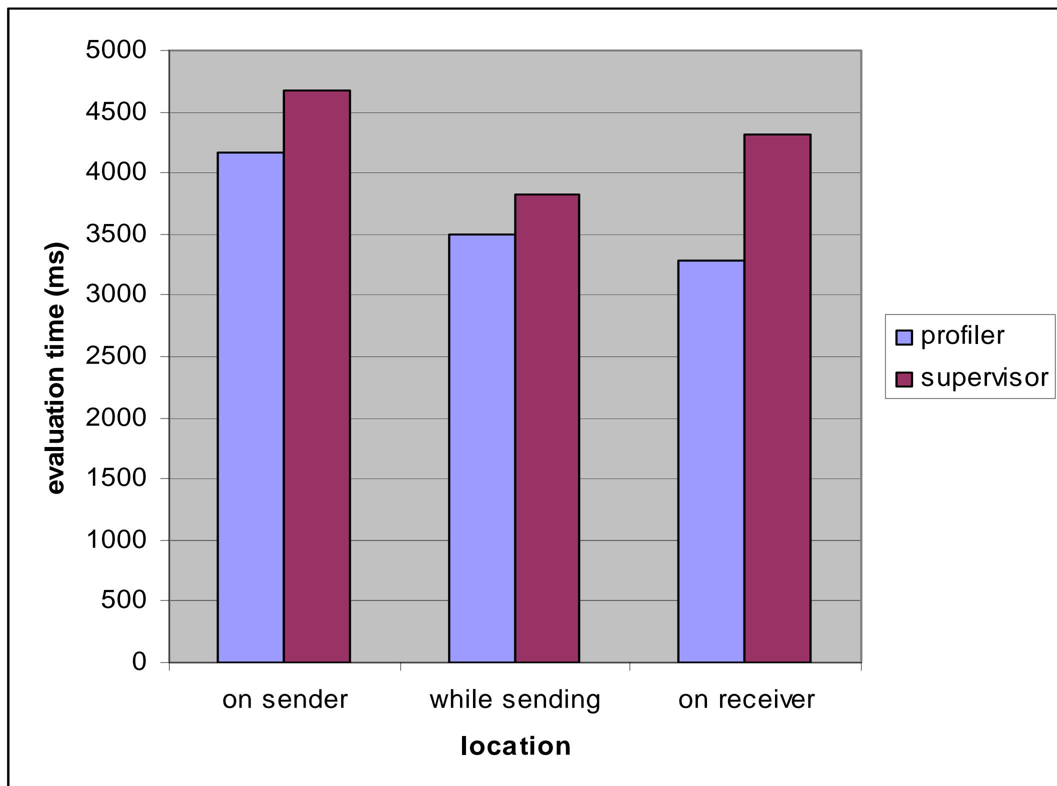

Figure 45: Average timing results to finish the application

4.5.4.2 Time Needed to Send the Components

Table 16 and Figure 46 shows the time needed to transfer our components with the different data rates. This experiment was executed only one time, so we do not show standard deviations. It seems that using a supervisor is beneficial for sending objects over the network, since they do not contain extra checking and migration code. It makes them smaller, and reduces the time needed to migrate them, even if we did transfer almost identical components in each case. Future experiments could show if this time we gain while sending could compensate for possible time loss when we use a supervisor to run the application (see Figure 45).

We can also observe that the time needed to send the objects seems almost identical for the normal sending and profiling strategy. This was something we expected, as our application was implemented to send objects of almost equal size.

Table 16: Time to send the components (ms)

data rate	2.4 kbps	9.6 kbps	56 kbps
normal	77483	24275	9337
profiler	78199	25041	9450
supervisor	53399	14941	3918

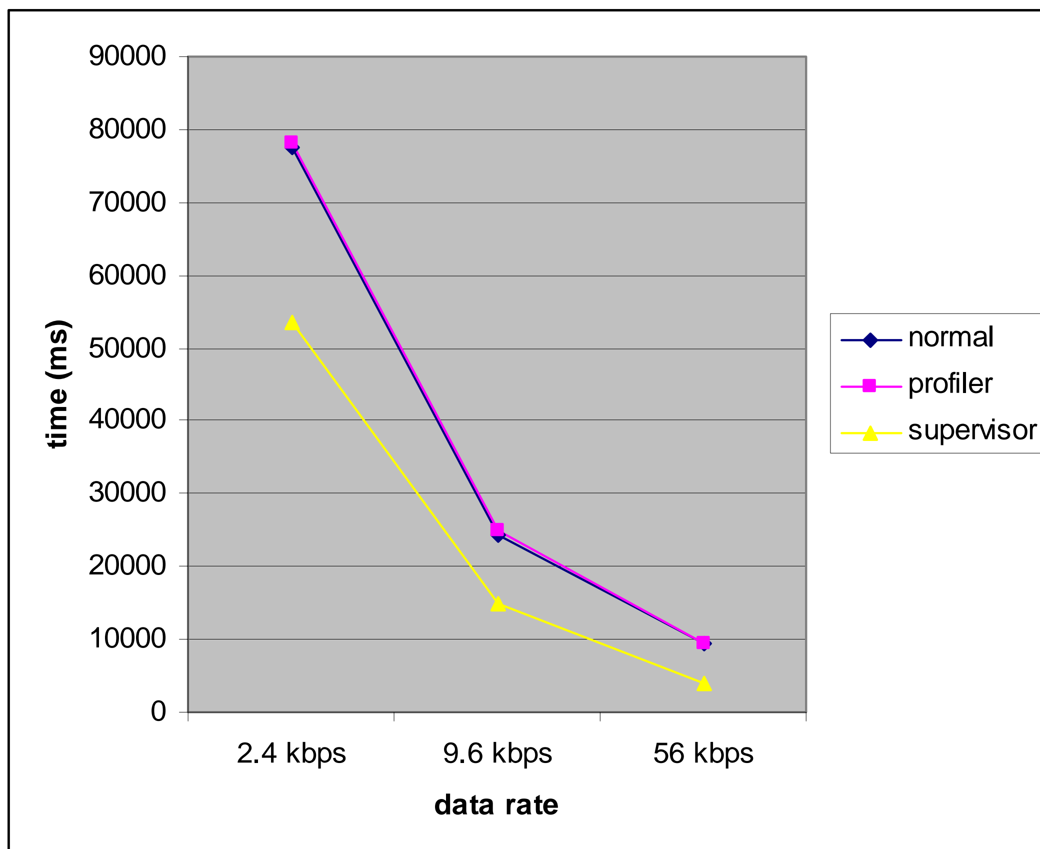

Figure 46: Time to send the components

4.5.4.3 Time Needed for the First Draw

The benefits of progressive mobility using component streams are best shown if we calculate the time needed to see the first drawn fractal after we started sending the components. This

can be seen in Table 17. This experiment was executed only one time, so we do not show standard deviations. With normal sending, we have to wait until the application is completely downloaded and started. This waiting time can be very long if low data rates are applied. When using progressive mobility using component streams however, we can see the first drawing after a few seconds, while the transfer is still busy. This drawing is located on the serving hosts, where the application was started, since we only have two components in our application. In this setup the availability of the application is not compromised by its migration.

This can demonstrate the use of the technique for hand-held devices. If an application needs to be transferred to a handheld device it can do so while it is still in use on the desktop computer, thereby generating immediate user feedback and minimizing user interface latency to a minimum.

Table 17: Time to first draw (ms)

data rate	2.4 kbps	9.6 kbps	56 kbps
normal	82806	29299	13736
profiler	3762	3844	3808
supervisor	3990	4193	3721

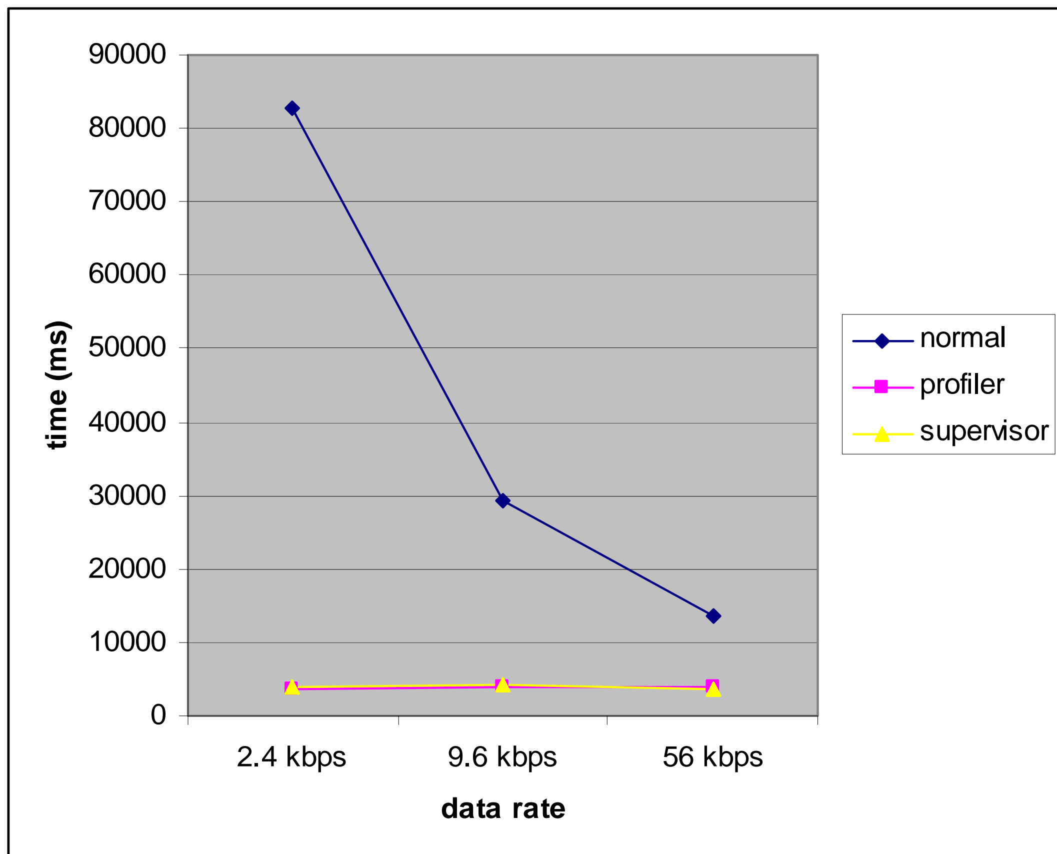

Figure 47: Time to first draw

In other applications, consisting of more components, we could send a small user interface component to the receiver first, so that the user will see the first results at the receiver after a shorter period. In our experiment, the displaying component was bigger, so it took most of the migration time.

The results on different strategies are compared in Figure 47. The graph represents the time to the first drawing using different data rates, and it shows that there is no latency while using progressive mobility using component streams. The time to wait is almost exactly the time to complete the application task, and this is the case for the profiler as well as the supervisor strategy.

If high data rates are applied, we will not see such a difference with normal sending of this rather small application. As components are almost immediately transferred and started at the receiver, using progressive mobility using component streams might even slow down the first drawing. This happens for example when the drawing component is interrupted on the sender before it finishes its task. If however we design the application so as to prevent the interruption of the drawing component at the sender we always become a first complete drawing very fast after the start of the migration even with high data rates. In general however, as the size of an application increases then scenario's as described above becomes possible even for higher data rates since the time of migration is defined by the size of the application and the data rate of the network. See also section 4.3.2.1: Component Migration Time.

As we noted before in the adapted gremlin application (section 3.8.4), here too, apparently small changes on the design level of the application sometimes suffice to get a more optimal behavior if applying progressive migration. Much of the behavior of an application, migrating using components streams, can be modeled by smart engineering the number and size of the components and the decision which component(s) should migrate first. We will revisit these engineering and refactoring strategies later in section 6.4.4.

4.5.4.4 Time Gained in Comparison to Normal Sending

In Table 18 and in Figure 48 we show the time we gain in this experiment using progressive mobility using component streams with low data rates. Since our application keeps running, and is available while being transferred, we have some time during migration that can be used to continue the evaluation of the application. In this specific experiment, it was possible to make several drawings on the sender before the application was completely transferred. Here too, this experiment was executed only one time, so we do not show standard deviations.

These extra drawings were only possible because of progressive mobility using component streams, since with normal sending, we would just be waiting. The results for this experiment should be interpreted as existential and does not reflect a universal rule. Even with the same application we will not benefit from this extra drawings on high data rates, since the size of the application to migrate is rather small the time gained will become insignificant in comparison to normal downloads. The migration time will become a lot shorter.

Also important is the fact that the application must be available for the user to interact with it, since the application started running on the serving hosts. This is not a problem when the sending host is a hand-held computer, or when we have a bigger application and we send a small user interface first.

Table 18: Time gained versus normal sending

data rate	2.4 kbps	9.6 kbps	56 kbps
profiler	79044	25454	9928
supervisor	78816	25106	10015

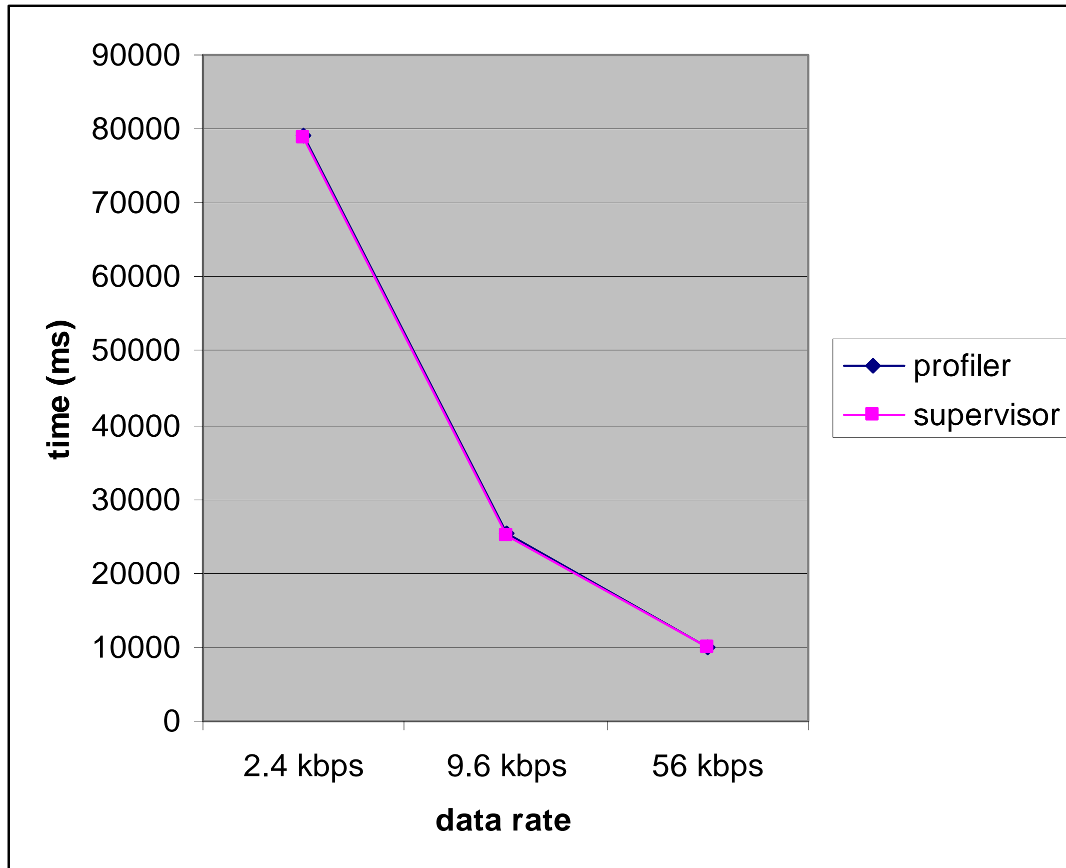

Figure 48: Time gained versus normal sending

4.5.5 Discussion

Java is not an optimal platform for implementing progressive mobility using component streams. It does not feature the perfect autonomous components we find in languages as Borg. Therefore Java components will need to delegate certain tasks. For example, if a Java component decides to migrate it can prepare this task for another component and then release control to enable that other component to do the job.

However, it seems that using a supervisor has some benefits for sending objects over the network, since they do not contain extra checking and migration code. It makes them smaller, and reduces the time needed to migrate them, even if we did transfer almost identical components in each case.

We can also observe that the time needed to send the objects seems almost identical for the sending and profiling strategy. This was something we expected, as our application was implemented to send objects of almost equal size.

The benefits of progressive mobility using component streams are best shown if we calculate the time needed to see the first drawn fractal after we started sending the components. This

can be seen in table 4.3. With normal sending, we have to wait until the application is completely downloaded and started. This can be very long on low data rates. When using progressive mobility using component streams however, we can see the first drawing after a few seconds,

Our experiments showed how progressive mobility using component streams is important for low data rates environments. While no time is lost during migration on high data rates, a lot of evaluation time is gained applying low data rates. This time can be exploited by the user, since the application is still available. The user will also see faster results, and will not have to wait for the application to start finally. These experiments on a real, but simple application proved we could eliminate network latency completely. Of course there is still a question of scalability. Future experiments may show if the progressive mobility using component streams technique is also useful for bigger applications, which have more components and less predictable behavior.

Other experiments may even result in new interesting techniques. For example, we could send an interface of an application to a handheld device, and migrate the computation to it using progressive mobility using component streams. But we would keep the computation running on the server, which has more processing power, for as long as there is a network connection. We would only start the computation on the handheld device when the network connection fails. In this situation the application would run in the most efficient way during migration, and be able to adapt itself when a problem occurs on the network.

4.6 Experiment to Reduce System Latency by Parallel Evaluation

System latency can be reduced by exploiting the implicit parallelism of a network. In this section we describe an experiment where we split our application in two components and design the application so as to allow parallel evaluation of the components.

In this experiment we will not only exploit parallelism between the evaluation of a component and the migration of another component but also parallel evaluation of the two components, one still on the sending host the other already arrived at the receiving host. This parallel evaluation is started immediately after the migration of the first component,

We report on experiments in the Smalltalk environment to implement component streams [Evenepoel 2003] and the exploitation of parallel evaluation. We applied the same strategy as in the previous section: fixed migration strategy under control of a supervisor since this environment does not support autonomously migration either. In this setup we transformed the design of the application in order to achieve concurrent processing

4.6.1 Process Migration with Opentalk

According to the Opentalk manual, all objects can be sent over a network by value using the asPassedByValue method. Unfortunately we discovered an important exception to this claim: Process-objects.

Exporting a process with Opentalk by reference is straightforward, and easy to implement. A process could be suspended, exported to a remote Smalltalk image, and resumed, as long as the connection to the source node was kept alive. But if we want to migrate a running application, the references to the source node must be cut off one time or another. So the processes should be passed by value, or the process' instance variables should be passed by value. A deficiency in the current design of Opentalk prevented that, apparently because the suspended context of the process could not be passed by value. We also found that we could

not export a process that was created in a workspace (an editable TextWindow), because it is bound to an empty namespace.

4.6.2 Using BOSS

If it is not possible to transport a Process-object by value, we could not implement process migration and application streams in Opentalk. We discovered however that Process instances could be transported by value under certain circumstances by deploying the Binary Object Streaming Service (BOSS), to pack objects in a binary format.

BOSS seemed to be capable of packing a (running) process. If we transport the BOSSed process to another image on a different host, we can resume the process there. First, we pack the process into a binary file, transport the file via File Transfer Protocol (FTP), unpack the file at the remote location, and resume the process.

The problem here was that we used the FTP-mechanism of the underlying operating system instead of the Opentalk environment. We know that BOSS first writes its data to a Stream, so we can also capture the BOSS-Stream and transport it via Opentalk to another host, where the BOSS-Stream is unpacked in a process-object.

4.6.3 Limitations in Current Smalltalk Environment

When we were trying to stream applications in Smalltalk, we discovered a number of limitations in Opentalk and BOSS when working with Process-objects. We managed however to develop some workarounds. We describe some of the lesser known problems in Smalltalk related to process migration:

4.6.3.1 Passing by Value

When we wanted to pass a process-object by value to another host, an error message “A Primitive has failed” is generated. This was caused by the fact that the Process instance variable suspended context could not be passed by value to another image. We found a workaround for this problem by packing/unpacking the process-object with BOSS, and passing this BOSS object by value to another image. As an extra advantage we noticed that sending BOSSed objects was on the average four times faster than just relying on the Opentalk object marshalling.

4.6.3.2 Order of Object Instantiation

It is not possible to create the necessary communication objects (e.g. RequestBroker, remote objects ...), before creating the BOSSed process. If this is the case, the transported BOSS-stream will fail to unpack the process-object at the remote location. After a while, a “Remote Invocation Timeout”-exception will be generated at the sender side. The workaround is to make sure that the process is BOSSed before the communication objects are set in place.

4.6.3.3 GraphicsHandles Cannot be Stored by BOSS

If the process to be BOSSed has references to graphical containers the following error message is generated: “GraphicsHandles cannot be stored by BOSS”.

Since the fractal experiment that we want to implement in Smalltalk contains a window we where forced to create the graphical containers externally and draw on these containers by sending draw-messages to a reference of these containers or by painting the fractal on any active VisualWorks Window at the remote host.

4.6.4 Experiment setup

In order to be able to compare the different environments we implemented the same fractal drawing application as in the previous experiment using the workarounds described above to cope with the current Smalltalk limitations.

Again the application becomes divided in two processes: the CalcProcess responsible for calculating the fractal and DrawProcess responsible for drawing it. We consider the strategies:

4.6.4.1 Calculate and draw the fractal, migrate afterwards

This strategy would be beneficial when the sending host has a lot of computational power, while the remote host lacks this power. An example of this setup is a computer connected to a PDA: the computer can calculate the fractal and send the resulting fractal drawing to the PDA, minimizing the use of the PDA's reduced computational power.

4.6.4.2 Start CalcProcess and migrate the DrawProcess

This strategy is appropriate when a user is waiting for the results of the DrawProcess at the remote host; the DrawProcess can start drawing the fractal immediately after migration, showing the user already a part of the generated image. This reduces the user interface latency.

4.6.4.3 Migrate the CalcProcess first, then the DrawProcess

If the remote host has more computational power than the local host, the CalcProcess finishes its computations faster, and the DrawProcess will have to spend less time waiting for new data. This way, the total evaluation time of the application is reduced.

4.6.5 Implementation

We implement the ***Start CalcProces, and migrate the DrawProcess*** (section 4.6.4.2) strategy because this strategy allows parallelism of the component evaluation. While the CalcProces is evaluation on the sender the Drawprocess is migrated to the receiver. At arrival its evaluation can start immediately and it will be able to draw the results calculated at the sender. The draw and calculate components can now run in parallel if appropriate. Later, the Calcproces at his turn can migrate to the receiver and continue his calculations there, this time on the same host as the Drawprocess. We compare it with the ***Calculate and draw the fractal, migrate afterwards*** strategy where no parallel processing is applied.[15]

4.6.5.1 Strategy without parallel processing

The application is composed of four loosely coupled components: CalcProcess, DrawProcess, SupervisorProcess and a SharedQueue. This example is an example of a combination of self triggered migration and the supervisor strategy. The processes CalcProcess and DrawProcess trigger their own migration by letting the SupervisorProcess know that they are ready to be migrated. Then the SupervisorProcess migrates the specific processes to the remote host. This technique has as advantage that the processes do not have to wait actively, occupying the processor, until the supervisor grants them the permission to migrate. The fourth component is a SharedQueue, this provides the communication channel between CalcProcess and DrawProcess: the Calc Process puts, after calculating a line segment and color, the data on the

[15] There were two computers involved in the experiment, one Pentium II 266 Mhz Celeron 192 MB RAM, with Mandrake Linux 8.1 installed on, and one Pentium II 266 Mhz Intel 192 MB RAM, with Windows '98 as the operating system. The computers are connected in a 10 MBit network.

SharedQueue, and the DrawProcess gathers these data to complete the drawing of the fractal. When the application is started, all processes are started simultaneously; the SupervisorProcess however runs at a slighty lower priority and waits for a signal from the other processes. The CalcProcess and DrawProcess both having a slightly higher priority, and signal the SupervisorProcess when they want to be migrated, after their task is finished. Once the SupervisorProcess got both signals, it migrates the processes and the SharedQueue to the remote host.

4.6.5.2 Strategy with parallel processing

This strategy applies the same component structure as the previous one but now the DrawProcess is migrated immediately to the remote host, after it signaled the SupervisorProcess it wanted to migrate. While this component is migrating, the CalcProcess is calculating the fractal's line segments and sends them to the SharedQueue. When the DrawProcess has finished migrating, it starts collecting the data from the SharedQueue, and starts drawing the line segments on a Window at the remote host. When the CalcProcess has finished calculating, it signals the SupervisorProcess it needs to migrate too. Then the SupervisorProcess migrates the CalcProcess and SharedQueue to the remote host. Since in this strategy the CalcProcess and the DrawProcess are allowed to run in parallel on different processors we expect a decrease of the total evaluation time of the application.

4.6.5.3 Results

We ran the first experiment 50 times, and calculated the average time it took to calculate and draw the fractal and migrate the components afterwards.

The average time needed was **21 sec 397 ms** with a standard deviation of 2 sec 56 ms.

We ran the second experiment also 50 times, and calculated the average time it took to complete the application. The average time for this experiment was **19 sec 637 ms**. with a standard deviation of 2 sec 16 ms.

4.6.6 Discussion

If the workload of an application could be shared by two complementary processes running in parallel the total time needed would be half the original one.

The gain in time in our experiment (9.1 %) is less than this hypothetical maximum of 50%. One of the reasons is that we did not deploy a separate processor for the migration phase, this means that the Calcprocess can only start its activity after the Drawprocess is migrated.

4.7 Design Guidelines

4.7.1 Necessary Conditions for Removing Network Latency

In order to be able to move every component in parallel with the evaluation of the application independent of the chosen migration strategy, we identified a number of conditions that must be satisfied (Table 13). If all these conditions are satisfied it suffices to migrate the components at the point of time where their largest idle period starts.

If we build a new application these design rules should be kept in mind. It will not always be possible to comply with them completely, but the more we approximate them the more the application will benefit from the proposed technique. If we need to stream an existing application, we may need to adapt it to comply better with the above conditions.

If the first condition is not met the technique can still be deployed but migration of the application will then cause some delay in its evaluation. We expect however that in many cases architectural transformations could be applied to transform the original application to an equivalent one that complies better with the first condition.

If the second condition is not met the migration of the application will also cause some delay in its evaluation. If the exact onset of the idle time is not known in advance it will be possible in some cases to estimate the delay based on statistics obtained from application profiling. Modifying the application at its design level could transform the original application to an equivalent one that complies better with the second condition.

If the third condition is not met the migration can only be optimized for one of the conflicting components although here also architectural transformations at the design level may resolve the conflict.

4.7.2 Guidelines

Based on the above conditions, we will now suggest a number of guidelines for building applications that are able to stream efficiently in an environment where efficiency, availability and fast migration are important.

- **Autonomous components**

In order to obtain components that are able to migrate independent from another host without the creation of extra inter-component communications infrastructure, the components should be able to communicate transparently with each other without knowing or even bother where their partners reside. Moreover the components should always communicate in an asynchronous fashion. An autonomous component should be designed as a separate entity, sending messages and receiving messages from other components, not as an entity which transfers its control flow to other components [Van Belle et al. 2001].

- **Numerous components**

If the number of components increases so will the idle time per component. Architectural refactorings could be applied as an optimization technique to increase the number of components without affecting the applications behavior.

- **No monopolizing components**

Equal sharing of the workload over all components is the most ideal situation to allow each component to migrate during its idle time. This equal distribution of the workload is only possible in theory. However in practice it suffices that the component that needs most of the time, has a workload that is smaller than the sum of the workloads of the other components.

- **Strong mobility**

The underlying framework should support strong mobility. If a component migrates during the evaluation of the application it must be able to migrate including its computational state and runtime stack.

- **Separate processors**

If one needs to eliminate the network latency completely the migration of the components should be performed by a different processor than the one that evaluates the application itself. The ideal, but mostly unpractical setup is to run each component on a different processor, eventually a specialized co-processor.

- **Intelligent supervisor**

The supervisor needs to decide when which component has to move thereby freeing the components themselves from this task. These rules can range from simple static rules to

dynamic rules that change under control of the supervisor who reasons over the dynamics of the running program, over its rules and itself.

4.8 Summary and Conclusion

New network architectures in ambient intelligence environments where connections between partners are no longer predictable need new solutions to support the inherent dynamics of these environments.

Mobile code is a plausible candidate to ensure the connection between different moving software components or devices but it will not always be possible to migrate this code as a whole, since the connection time between, possible moving objects, is not predictable.

Ubiquitous communication in ambient intelligence environments only makes sense if the objects that compose the network are always available to respond to the requests of other objects even during their unpredictable migration time. Since connections between hosts in these new environments are more volatile than in static networks there is the need for mechanisms to split up the code in smaller parts that will fit in the limited timeframes in order to migrate the parts of the code progressively in time to other objects.

To keep the application available during its migration we will also need to put systems in place that allows the code to continue its evaluation during the progressive migration so that the application remains available for users and other applications at all time.

Precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately usable (ready for evaluation) at the receiver's end.

In this second theme, we have built a proof of concept of a system that breaks up the code of an application in smaller parts and sent them one by one progressively in time, while the evaluation of the application continue.

Performance of an application is most commonly measured by *overall program evaluation time* and network performance is most commonly measured in *network latency* but in a mobile environment performance is also measured by *application availability, invocation latency*, and *user interface latency*.

Overall program evaluation time is the time between the invocation of an application and the end of the evaluation of the last instruction.

Application availability is the inverse of the time an application "freezes" during migration.

Network latency is the time the application needs to travel over the network.

Invocation latency is the time from application invocation to when evaluation of the program actually begins.

User interface latency is the time a user has to wait between his demand and a user interface reaction of the system.

Table 19 gives an indication of the performance of progressive mobility using components streams in these different domains.

Table 19: Properties of the Component Stream Technique

	Overall program evaluation time	Application availability	Network latency	Invocation latency	User interface latency
Component streams	-	+++	+++	+	++

To conclude this chapter we discuss these results in the most important dimensions of the conceptual framework provided in chapter 2.

4.8.1 Network

We expected that **network latency** could be hidden by moving each component during its idle time, this expectation is confirmed by all our experiments, network latency was hidden completely.

Invocation latency and **user interface latency** could be reduced by starting up the application at the sending host at the same time its migration is triggered (section 4.5.4.2).

It is also important to keep in mind that in an ambient intelligence environment two kind of networks must be considered, connection-oriented networks and connectionless networks.

If application availability is the main goal to implement component streams the setup time in **connection-oriented networks** is not so important since the application will remain available at al times even if its migration is delayed somewhat. However, if the application is designed to take advantage of the parallelism of its components then the initial setup time will delay this parallelism.

In the case we deploy components streams with the goal to reduce user interface latency then the setup time will play a major role and may have a great influence on the users perception of the application.

Connectionless Networks are also the ideal environment to implement progressive mobility using components streams. Compared to the connection-oriented networks the packets are typically a little larger since they need to contain the complete address of the receiver but in practice the size of this extra data can be ignored compared to the size of the actual block of data sent.

If there is no setup time then one might also consider parallel file transfer to send more than one component in parallel to the receiver. This might be especially useful if some components exhibit a high degree of coupling between them. By sending these components together in parallel, one can avoid the heavy traffic over the network between these components while they become temporarily distributed.

4.8.2 Application

Applying component streams excels in application availability but in general we expect the overall program evaluation time to increase since the application becomes distributed during the streaming phase. In some cases however we might exploit the temporary parallelism to compensate for the delay introduced by the distribution and for small applications in some test environment as in ours we might even see a decrease in evaluation time.

We showed that application availability could be enhanced by continue the evaluation of a component on the sender while a copy of its code is migrated in parallel to the receiver (section 4.5.4.3).

Besides a proof of concept in Borg we provided some existential examples in other programming environments as Smalltalk and Java in order to show that for these applications the technique is useful.

We investigated in different designs of a fractal draw application so as to determine the possibility to reduce system latency by introducing parallel component evaluation.

System latency could be reduced by starting up an application before the application is completely migrated (section 4.5.4.1) and by applying parallel component evaluation (section 4.6.5.2).

We noted the importance of the design of the application in function of the migration strategy. In this context, we provided some design guidelines.

4.8.3 Techniques

We explored three different possibilities to harness parallelism. We experimented with the same kind of parallelism as in chapter 3, starting the evaluation of application parts already received while the rest of the application still needs to migrate. But since the application is already running before its migration we also exploited the parallelism of the migration of copies of components that still are running on the sender and the parallelism between components still running on the sender and components already arrived and running on the receiver.

In this chapter we migrated running applications so we made use of the technique of strong mobility.

The determination of the universal nature of the techniques or the demarcation of the domain in which the technique proves useful is left for future work.

5 Progressive Anticipative Mobility using Proactive Migration

He who would travel happily must travel light
-- Antoine de Saint-Exupery (1900 - 1944)

5.1 Abstract

The migration of code in ambient intelligence is needed to provide support for the dynamic character of these environments but at the same time this is not a trivial process. Ambient intelligent networks are more volatile than static networks, which makes the timeframe available for the migration unpredictable.

Therefore we need some kind of mechanism to break up code into smaller parts and send them one by one to enhance the chance that it will fit in the current timeframe.

There is also the need for mechanisms that allows the code to continue its evaluation during the progressive migration so that the application remains available for users or other applications at all time.

We explored these mechanisms in the two previous chapters but we do realize that providing application availability by the mechanism of component streams comes with the price of reduced evaluation speed introduced by the temporarily distribution.

In this last theme we explore an alternative mechanism to hide network latency where we migrate the large bunch of the code in advance thereby taking advantage of possible surplus bandwidth that will not be exploited otherwise. When the real code migration is triggered then there is only the need to send the difference between the current state of the code and the code that was sent in advance. If this delta is small enough, the migration of its representation can be so fast, that for the users perspective the application remains available at all time.

In this chapter, where we provide a proof of concept of the theme of proactive migration, we start by describing the proposed technique and more specifically the technique to calculate the difference between computational states, here called the delta.

Then we describe an experiment to calculate the delta and discuss the expected results.

Roadmap:

- Introduction
- Proposed Technique
    - Basic Observations, assumptions and restrictions
    - Technique description
- Experiment to calculate the delta
- Results
- Discussion
    - Expected Gain
    - Applications without implicit stack operations
    - Dealing with Large Deltas
- Summary and Conclusion

5.2 Introduction

With the advent of *Ambient Intelligence* (AmI), mobile code will become an important medium to support this intelligent environment. Objects that do not move relatively with respect to each other can rely upon current communication protocols to provide a stable connection but the connection between moving objects poses new challenges.

Since connections between hosts in these new environments are more volatile than in static networks there is the need for mechanisms that allow the code to continue its evaluation during the migration so that the application remains available at all time.

The theme of streaming components, explored in the previous chapter, proposed such a mechanism but it also introduced extra latency since the applications became temporarily distributed.

In this third theme, it is our goal to build a proof of concept of a system that sends proactively the code to a potential receiving host so that most of the migration work can be done in advance thereby taking possibly advantage of surplus bandwidth in the network that would not be exploited otherwise. Then when the real migration is triggered we only need to send the delta between the computational state already sent and the new current computational state. If this delta is small then the migration of this delta and its adaptation to the already received code can be so fast that for the perception of the user the application remains available at al time.

Network latency is hidden since at real migration time we only need to send a block of code that is much smaller than the block of code we typically need to migrate.

The feasibility of the technique has been validated by implementing a prototype tools in the Borg mobile agent environment. A simple experiment in Borg, to measure the size of the delta, shows that it allows to migrate the application in 2% of their original migration time.

In this third theme, our contribution is the introduction of progressive, anticipative mobility using proactive migration.

Progressive anticipative mobility using proactive migration is proposed as a technique to compensate for network latency and to enhance application availability of migrating applications. In the same sense as progressive anticipative mobility using pre-fetching of permuted code and progressive mobility using component streams this technique applies progressive mobility. Progressive mobility potentially allows applications to migrate piece by piece what may result in early startup at the receiving host and possible smooth evaluation with limited or no disruption.

Progressive anticipative mobility using proactive migration is proposed as an optimization for *progressive mobility using component streams* [Stoops et al. 2003a, Stoops et al. 2003b]. The main property of progressive mobility using component streams is that it migrates running code. A possible disadvantage of the technique is that applications become temporarily distributed during the streaming phase which may slow down some types of applications. As we will show, proactive migration has the potential to avoid this temporary distribution, thereby allowing the migrating application to run almost continuously at full speed.

The technique presented here might be useful when we know in advance when we are going to migrate. In ambient intelligent environments by example, where hosts are moving in relation to each other, one may foresee that an application possible will migrate to a host that comes physically in the neighborhood. Shopping agents [Chavez 1997] often know in advance their migration path and even if they plan their itinerary dynamically they are often able to decide on the next host to visit before they start the actual work on the current host. Another possible application of mobile code is load balancing. Here too, the overload of a host can often be predicted as well as the host to flee to.

If we manage to send the bulk of the application in advance to the receiving host we can, when the real time to migrate has come, obtain sometimes a high-speed migration of our running application in a fraction of the time needed for normal migration. It potentially allows applications to migrate very fast from host to host without a significant loss of evaluation time during the migration phase.

Given the complexity of the implementation of this technique we start by building a proof of concept in Borg, an environment that stores the computational state of the application in one chunk of memory and by its nature facilitates the expression of mobile components.

We demonstrate the feasibility of the technique by migrating an application in the Borg mobile agent environment [Van Belle et al. 2001] in order to measure the difference in migration time. In our setup the technique proves to be useful, but the determination of a possible more universal nature of the technique is left for future work.

5.3 Proposed Technique

5.3.1 Basic Observations, assumptions and restrictions

As mentioned before, transporting mobile code over a network is in general the most time-consuming activity, and can lead to significant delays in the migration of the application. This is especially the case in low data rate environments such as the current wireless WAN communication systems or in overloaded networks. In a classic migration scheme, everything that needs to migrate is sent in one big block of data over the network.

Our approach of proactive migration allows us to send parts of the code in advance and therefore may gives us also the opportunity to take advantage of periods of low network traffic.

We assume two preconditions to apply proactive migration in an efficient way:

1. **The internal representation of the computational state is stored in one chunk of memory.**
2. **The internal representation of the program code remains constant.**

The Borg environment, used for our experiment, represents the application code in an **abstract syntax tree.** The computational state of an application is represented by a **stack** (for continuations and intermediate evaluation values) and a **dictionary** that contains the names of the variables and references to a block of memory where the values of these variables reside, called the **heap**. In the Borg environment these entities are contained in one chunk of memory. This will satisfy precondition 1.

The Borg environment also allows reflection. Reflection is the ability for a program to manipulate its code and computational state as data during its evaluation [Maes 1987]. The expressiveness of Borg allows an application to reason about itself and its computational state during its evaluation. In our first experiments we will allow all kind of reflective behavior except the adaptation of the abstract syntax tree. This will satisfy precondition 2.

Here too, we assume that the know-how and know-when of the migration of partitioned code is located in the sending host, so we apply a push strategy. However, this does not exclude the possibility of successful combinations with a pull-strategy.

The size of the application in our experiment in Borg, is so small that it falls outside the window (see section 2.1.6 Window of Opportunity - page 39) in which the transportation time to send a block of code is directly proportional with the size of the block of code. This makes it impossible to reduce invocation and user interface latency at the receiving host since the transportation time to send the complete block of code is the same as the transportation time to send even a small part of it.

However, it is our only goal to provide a proof of concept in this very specialized programming environment. For the proposed technique itself, we do not make any assumptions on the total size of the application, so we may expect it to scale to larger applications.

5.3.2 Technique Description

The technique of proactive migration applies a progressive migration scheme. The running application is split into two components: a snapshot of the complete application and the delta of the computational states after a certain time. Basically the technique is a five step process:

1. Take a **snapshot** of the running application, i.e. take a copy of the code and its computational state, on the sending host.
2. **Copy** the snapshot to the receiving host **while the original application continues to run.**
3. Once the copy has arrived at the receiving host (or later when the real migration is triggered) **halt** the original application.
4. Define the changes, called **the delta**, in the original application during the copy phase of the snapshot. This delta contains the changes in the computational state.
5. Migrate and **apply this delta** to the, already migrated, snapshot and resume its evaluation.

Since each application contains parts that remain constant, the size of the delta will always be smaller than the size of the complete snapshot. This is where we can gain in migration time. Suppose we know five seconds in advance that we're going to migrate. At that moment we capture the complete application including its computational state and forward it already to the receiving host. Then, five seconds later when we really want to migrate we identify the delta of the current computational state with the already sent computational state and only transmit this delta across the network.

As an example, Figure 49 shows a simple Borg factorial program.

Figure 50 shows a sequence diagram that illustrates the behavior of a classic strong migration of this example program.

```
fac(n): if (n=1,
            1,
            n*fac(n-1)
        )
fac(100)
```

Figure 49: Calculation of factorial (n) in Borg

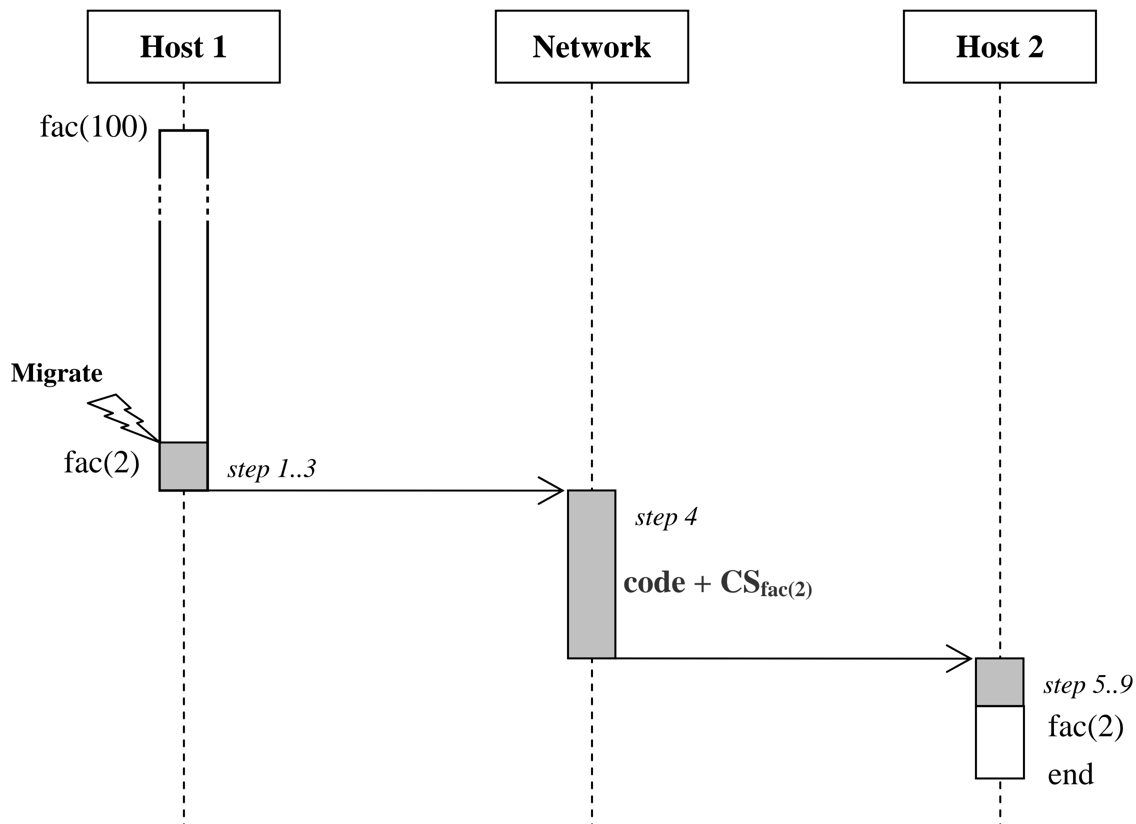

Figure 50: Sequence diagram – classic strong migration of a factorial calculation

We start by calculating fac(100). Suppose that the application receives an external trigger to migrate at the start of the recursive call with parameter n equal to 2. As a result of this trigger the first 3 steps of the classic migration scheme are launched on Host 1. If we call CS the computational state; then we transfer over the network during step 4: (code + $CS_{fac(2)}$). After the transfer, steps 5..9 of the migration scheme allow the application to resume on Host 2.

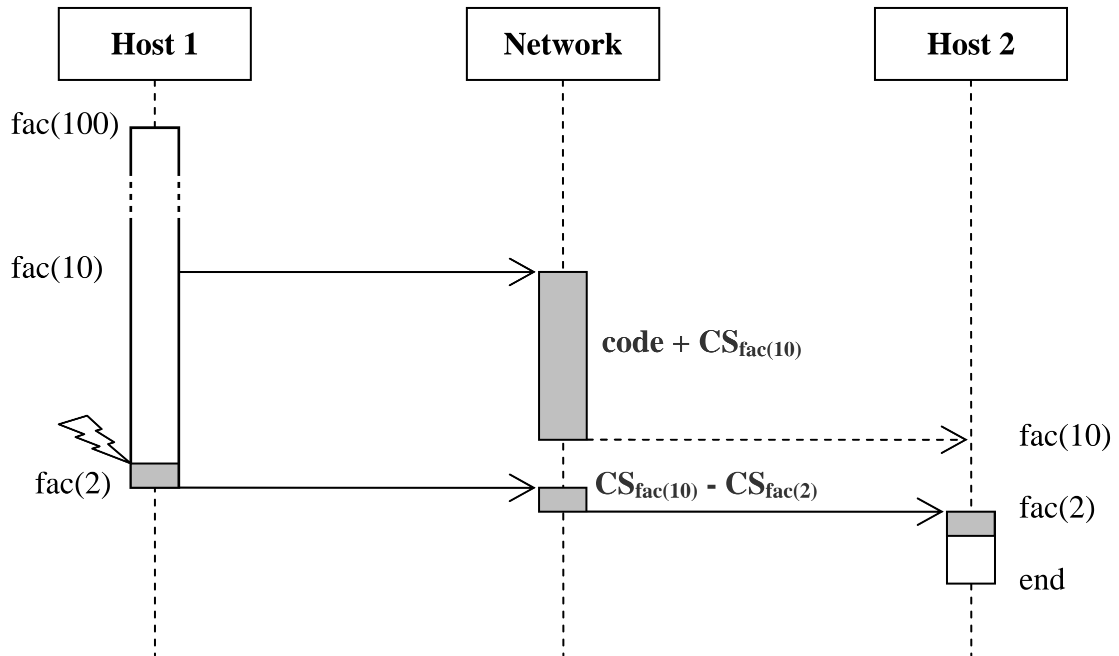

Figure 51: Sequence diagram - proactive migration of a factorial calculation

Figure 51 shows an example of proactive migration of the same factorial program:

We start again by calculating fac(100). At fac(10) we proactively migrate the bulk of the code and state but the application is not started on Host 2. As shown in the figure we assume that this migration process is able to run in parallel with the application itself. Then, at fac(2), we transfer the delta and resume the application at Host 2. We note that the migration of the bulk of data code + $CS_{fac(10)}$ takes much longer then the small delta $CS_{fac(10)}$ - $CS_{fac(2)}$. This allows the application, after the external trigger, to start up much sooner now at Host 2 resulting in a high-speed migration of the application.

The most challenging part of the technique is the identification, extraction, presentation, migration and reapplication of the delta. The reapplication of the delta can be seen as the reverse process of the extraction so we will focus on the identification and extraction part.

5.3.2.1 Computing the Delta

In order to identify the delta we need to compute the difference between two computational states. Reflection, available in Borg, allows us to capture the current computational state at any time in the Borg language itself. To calculate the delta between two states in an efficient way we add a native function: ***delta()*** to Borg. This function is written in C, the implementation language of Borg.

Taking a snapshot of the computational state is basically achieved by computing the transitive closure of all elements starting from the root of the stack and the dictionary. This closure will contain complete arrays, and objects pointed to by this root. In Borg this closure will also contain the abstract syntax tree. In the remainder of this paper we will refer to the Borg environment and our Borg prototype to explain the technique.

In order to explain the techniques to compute the delta in an efficient way we explore gradually more complex memory operations. As to simplify the operations somewhat we assume that the garbage collector is disabled between the capturing of two computational states. We show later how possibly to deal with garbage collection. We will now study consecutively:

- **Non-destructive memory operations**
- **Destructive memory operations**
- **Random access memory operations**

Non-destructive Memory Operations

Functional languages as Borg have the property that they never change the memory in a destructive way. Since there is no assignment operator available in a pure functional language the internal representation of the computational state will have certain properties that can be exploited to calculate the delta.

As in many garbage collected languages, the memory in Borg is allocated sequentially, as is the case with a stack data structure, and since we know that no memory content will be overwritten, the difference in computational states will be the newly allocated memory. If we compare the two computational states CS_1 and CS_2 (Figure 52) then the delta will exist of all memory allocated after the first state capture. For easy identification of this memory block we add a watermark at the end of the CS_1 memory block. This allows the serializer (step 2 of the migration process) to identify the newly allocated memory as the memory after the watermark.

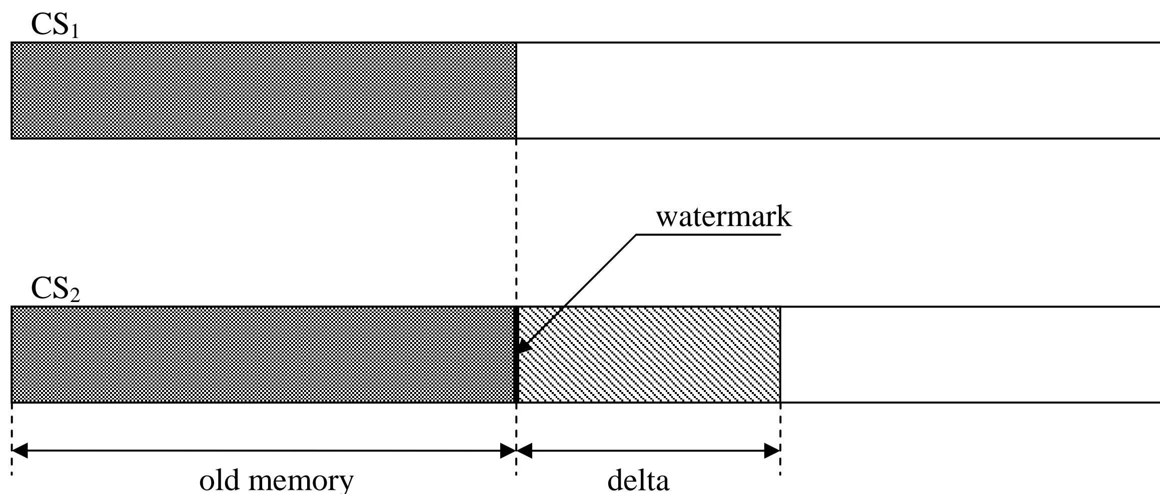

Figure 52: Watermark for delta definition in non-destructive memory

This technique is also used in an optimization technique known as generation scavenging [Ungar 1984] for garbage collection. Indeed garbage collection and serialization are similar operations. Both need to make a transitive closure. Only the serializer flattens this closure,

while the garbage collector compacts it. The generation scavenging technique is usable by both.

The old memory has been transferred to the receiving host, and when the new computational state has references to the old one (Figure 53), the pointers in memory must be adapted to point to correct addresses on the receiving host. This adaptation takes place during step 8 (Table 1 page 26) of the migration process.

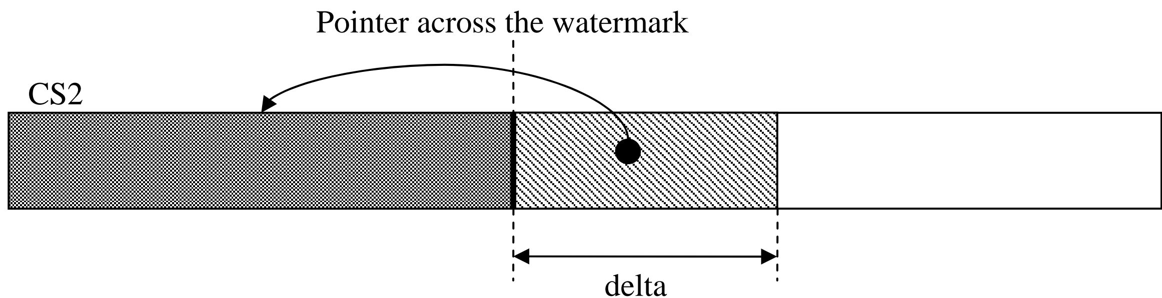

Figure 53: Possible pointers in non-destructive memory

Destructive Memory Operations

Unlike the functional language Borg, imperative languages allow destructive changes in memory so the internal representation memory block can contain pointers in both directions (Figure 54). Even functional languages will sometimes use invisible destructive operations in memory for performance issues.

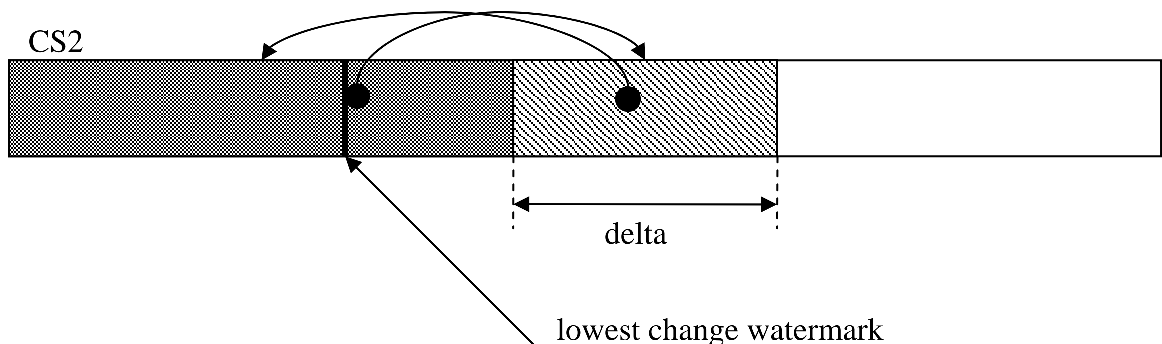

Figure 54: Possible pointers and watermark in destructive memory

In this case, we will need to compare both memory blocks byte by byte to determine all the differences. Fortunately a stack structure, heavily used by modern compilers [Grune 2000], is a friendly structure, it has no random access and once we find the 'lowest' change, we can be sure that everything above this point has changed too. Putting a watermark on the stack at this point makes it easy for the serializer to determine the part of memory to serialize.

Random Access Operations

We have discussed how to calculate the delta in non-destructive and destructive memory environments in a stack data structure. We will now discuss how to handle migration and subtraction of computational states in random-access data structures. The dictionary with the variable bindings and the values in the heap in Borg are typical random-access data structures.

When we wish to migrate a random-access structure we must explicitly keep track of the memory involved and changed. How to do this will be largely implementation dependent, but habitually memory can only be accessed through existing variables, so keeping a list of changed variables is often an option. Another alternative involves maintaining an array in which we explicitly keep track of all changed structures.

We could space-optimize this further by using dirty-bits, bits that flag a change of a memory word. In a garbage-collected programming environment we may find that some bits are already reserved for this kind of operation [Wilson 1992]. Under the presumption that we won't garbage collect we can reuse some of the spare garbage collector bits as dirty bits. This restriction isn't as harsh as it seems, if a garbage collect were to trigger, we could just calculate the delta and migrate early. Also, because a garbage collection and a serialization or very similar processes, we could trigger a garbage collect together with the first migration.

5.4 Experiment to Calculate the Delta

Figure 55 shows the Borg code used to calculate the size of the streams over the network. We declare an array ***a*** of size 100 and then declare and run an instrumented factorial function: ***fac()*** that fills up the array with the consecutive computational states. The factorial function is instrumented with the native function call***: call(cont)*** that returns the complete current computational state.

Then we apply the new native function ***delta()*** to calculate the delta ***d*** between $CS_{fac(10)}$ and $CS_{fac(2)}$. This function is written in C, the implementation language of Borg to be able to calculate the difference of two Borg computational states in an efficient way.

Finally we display the sizes of the serialized instances of the computational states $CS_{fac(10)}$, $CS_{fac(2)}$ and the size of the delta between them.

```
a[100]:0;
fac(n):if(n<2,n,{a[n]:=call(cont); n*fac(n-1)});
fac(100);
d:delta(a[10], a[2]);
display(size(serialize(a[10])));
display(size(serialize(a[2])));
display(size(serialize(d)));

>: Size of serialized stream: 25664 bytes.
>: Size of serialized stream: 25408 bytes.
>: Size of serialized stream: 515 bytes.
```

Figure 55: Borg calculation computational states

5.5 Results

In this factorial experiment we note that the delta between the computational states contains 515 bytes while the original computational state, including the abstract syntax tree was 25664 bytes.

For this small factorial calculation example the reduction in size is 515 / 25664 = 2% of the original stream size or a compression ratio of: 98%.

If we envision a setup in a network where the migration time is proportional to the size of the application (section 2.1.4.2), then the reduction in size will also lead to a reduction in migration time of 2% of the original migration time.

5.6 Discussion

5.6.1 Gain for the Factorial Example

If we run the factorial program (Figure 49) and send our first snapshot at fac(10) and then really migrate at fac(2) we can compare the stack and dictionary at those moments (Table 20).

Table 20: Stack and Dictionary Values during Evaluation Factorial Program

Stack	Dictionary		
fac(100)	n	100	
fac(99)	n	99	
fac(98)	n	98	
fac(97)	n	97	
...	...	...	
fac(12)	n	12	
fac(11)	n	11	
fac(10)	n	10	*proactive migration*
fac(9)	n	9	
fac(8)	n	8	
fac(7)	n	7	
fac(6)	n	6	
fac(5)	n	5	
fac(4)	n	4	
fac(3)	n	3	
fac(2)	n	2	*actual migration*

Migration of the full stack and dictionary at fac(2) would consist of 297 entries: 99 stack, and 99 name/value pairs. When we migrate at fac(10) we would have to transfer 89 * 3 entries. But if we then migrate at fac(2) and only need to serialize the entries created between fac(10) and fac(2), this would be only 9 * 3 = 27 entries. This leads to a size compression ratio of $\frac{27}{296} \approx 9\%$.

This gain in time can be attained if we only take the delta between the stacks and dictionaries in account. In reality we also need to include the abstract syntax tree in our first snapshot which will increase our gain in time even more.

5.6.2 Applications without Implicit Stack Operations

So far we have only looked at the factorial example. Such functions with implicit stack operations aren't very common in practice. In practice we expect more sequential function calls. So how does this affect the delta calculation? We will not to generalize but in this case of very small processes it might be favorably. Consider the program in Figure 56.

```
main():
{ fun100(); ...; fun10(); ...; fun2();
}
```

Figure 56: application without implicit stack operations

We start by calling fun100(), anticipative migrate at fun10() and migrate the delta at fun2().

With each function call the stack expands. However after each function call, it shrinks again. Therefore, at fun2(), we will have a much smaller stack as compared to the growing stack in the previous factorial example. In fact it will only contain the data for the main function and for the fun2() function. This implies that our delta will only contain the fun2() data. Compared with the bulk of data anticipatively sent, this delta is so small that very large time compressions may be expected, although there might be a possible performance loss because of the extra delta calculation.

If the stack is very large at the time of proactive migration and at the real migration time then the difference might be very small too. However, (1) we always need to proactively migrate the large stack, and (2) the difference between two stacks will never exceeds the size of the larger of the two, which will favor applications with little or no implicit stack operations.

A typical program will not behave like either of the two presented examples but will most likely have a performance situated somewhere between these two extremes. This claim can be supported by the fact that a typical program stack does not become very big.

5.6.3 Hardware Support

The technique of proactive migration has the potential to hide network latency and reduce system latency at the moment that migration is triggered but possibly introduces extra latency at other moments in time. The most time consuming actions might be:

- Taking a **snapshot** of the application, possibly more than once
- **Migrating** a copy of the snapshot, possibly more than once, to the receiving host
- Defining the **delta**

If the delays introduced by these actions are not acceptable then in some cases extra hardware support could avoid some of these delays.

One can imagine a dedicated *shadow* memory of the same size of the actual memory in use and connected with the *main* memory in order to allow **snapshots** to be taken in one machine cycle.

The **migration** of the snapshot to the receiving host could be handled by a second parallel processor to avoid delays in the original application.

If a shadow memory is in place, a bitwise XOR could define the **delta** also in one machine cycle.

5.6.4 Dealing with Large Deltas

Sometimes the size of a running application can increase a lot at a given point in evaluation time, e.g. the initialization of the Java user interface object as was the case in section 4.5.3. If we can foresee this increase in size it would be very beneficial to migrate the application before this point in time but under certain circumstances it is even possible to cope with these application enlargements afterwards.

The program in Figure 57 shows an example of code that potentially can generate a large delta.

```
main():
{...;
 fun: my_arr = allocate(10000);
 fill_up (my_arr);
...;
}
```

Figure 57: Application with a potential large delta

Suppose we migrate a first time right before the allocate and fill_up function call and transmit the delta right after it. Then the first transmission will be reasonably fast, but the second transmission will be a lot slower because the big chunk of new allocated memory filled up with calculated or retrieved data has to come with it. As a result there might be nearly no performance gain by applying proactive migration (Figure 58).

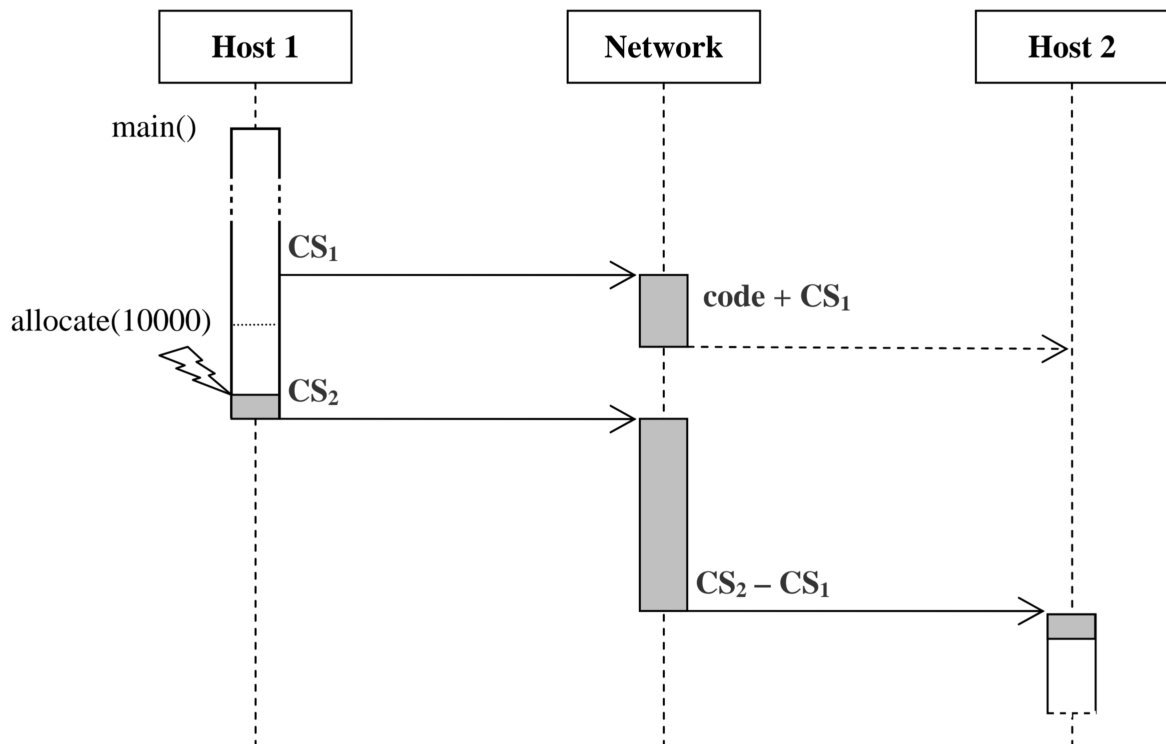

Figure 58: Large delta

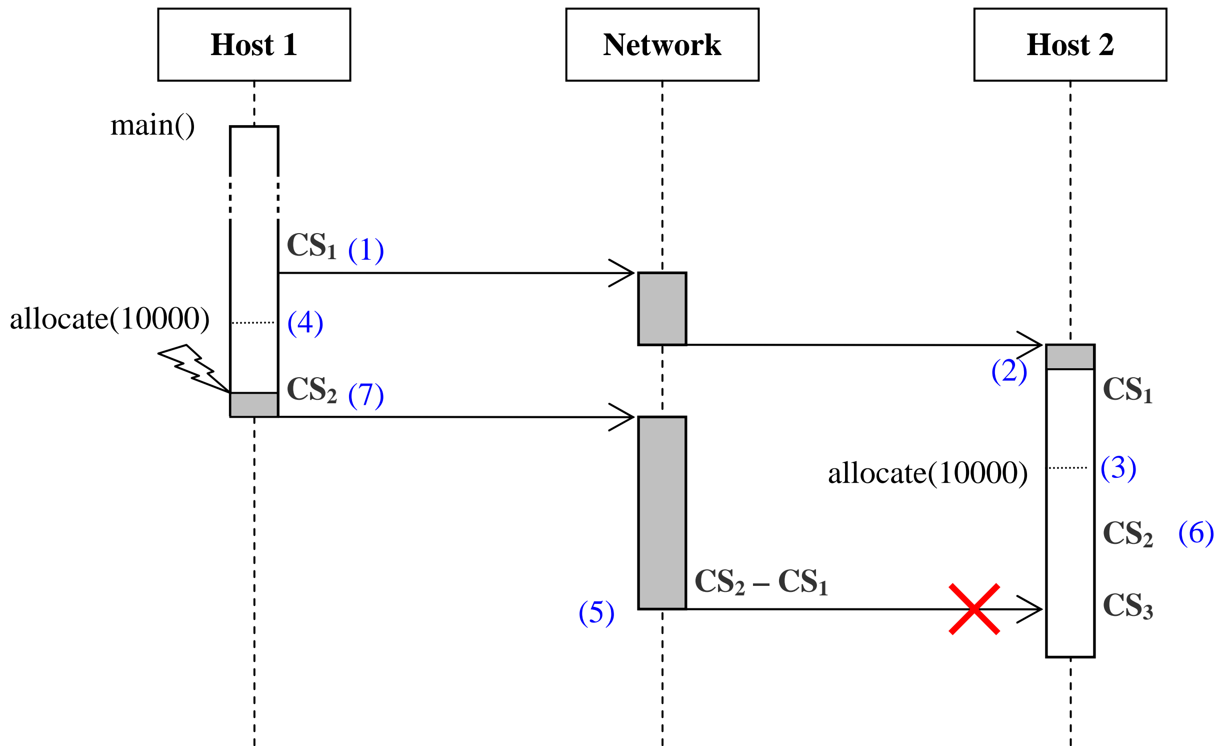

Figure 59: Dealing with a large delta

However, in the case that there are no interactions with external entities between the anticipative and the final migration and the application has a deterministic behavior as in our factorial example, there is something we can exploit. If we start the evaluation of the anticipative application immediately after arrival then the allocation of the new memory block can also take place at the Host 2 instead of only at Host 1. This scenario is depicted in Figure 59 end described below.

When the application is anticipatively migrated to Host 2 at computational state CS_1, (1) nothing prevents this host from starting the application already (2) as shown in Figure 59. So we can allocate the huge chunk of memory on the Host 2 (3) and continue execution there. The equivalent chunk of memory, allocated on Host 1 at (4) and send as part of the difference between CS_2 and CS_1, will finally have crossed the network possibly at a time (5). At this time the computational state on the receiving host CS3 has already surpassed the computational state CS_2 (6), the same state as the one captured on the sending host (7).

Now we might detect that the application at Host 2 has spend more clock cycles than the original one at Host 1 since the last anticipative migration (1). In that case, it would be of no use to replace the current computational state CS3 with the superseded state CS2 obtained from the original application and therefore we may ignore the received state and continue our own thread of evaluation at Host 2.

If Host 2 has no other tasks than to wait for the application of Host 1 we could choose to start computation always at the receiving host immediately after the anticipative load. Then if the sending host detects the big overhead of the second transmission, and there where no interactions with external entities since the proactive migration, it can discard the transmission and avoid the second migration completely.

If we adhere to the classic definition of *network latency* as: the time between the actual sending of an application and its startup at the receiver we notice that in these cases the application is started before the *actual* sending took place, so we obtain a **negative** network latency. We suggest a more general strategy of this approach in future work (section 6.4.5).

5.7 Summary and Conclusion

Mobile code will become an important medium to support an ambient intelligence environment. Objects that do not move relatively with respect to each other can rely upon current communication protocols to provide a stable connection but the connection between moving objects poses new challenges.

The theme of streaming components, explored in the previous chapter, proposed a mechanism that allows the code to continue its evaluation during the migration so that the application remains available at all time but it also introduced extra latency since the applications became temporarily distributed.

In this third theme, we provided a proof of concept of a system that sends proactively the code to a potential receiving host so that most of the migration work can be done in advance thereby taking possibly advantage of surplus bandwidth in the network that would not be exploited otherwise. Then when the real migration is triggered we only need to send the delta between the computational state already sent and the new current computational state. If this delta is small then the migration of this delta and its adaptation to the already received code can be so fast that for the perception of the user the application remains available at al time.

Performance of an application is most commonly measured by *overall program evaluation time* and network performance is most commonly measured in *network latency* but in a mobile environment performance is also measured by *application availability, invocation latency*, and *user interface latency*.

Overall program evaluation time is the time between the invocation of an application and the end of the evaluation of the last instruction.

Application availability is the inverse of the time an application “freezes” during migration.

Network latency is the time the application needs to travel over the network.

Invocation latency is the time from application invocation to when evaluation of the program actually begins.

User interface latency is the time a user has to wait between his demand and a user interface reaction of the system.

Table 21 gives an indication of the performance of progressive mobility using proactive migration in these different domains.

Table 21: Properties of the Proactive Migration Technique

	Overall program evaluation time	Application availability	Network latency	Invocation latency	User interface latency
Proactive migration	+	++	++	++	+

To conclude this chapter we discuss these results in the most important dimensions of the conceptual framework provided in chapter 2.

5.7.1 Network

Proactive migration excels in reducing the network latency especially for the transmission of mobile applications over a slow network. The technique is based on the observation that we can calculate the delta between two computational states and that this delta is always smaller than the original computational state.

Network latency is hidden since at real migration time we only need to send a block of code that is much smaller than the block of code we typically need to migrate.

In an ambient intelligence environment two kind of networks must be considered, connection-oriented networks and connectionless networks.

A large setup time in **connection-oriented networks** is not really a problem for applying this technique. The proactive migration can be launched even several times in advance during periods when extra bandwidth is available or when the connection time is guaranteed for longer time intervals. The exact point in time for this proactive migration is not important, so a possible large setup time will not influence this migration.

The real migration should be happen as soon as possible after the external trigger but even there is the opportunity to let the application continue its evaluation until the connection is setup. After the connection is setup, there will be still time enough to calculate the delta en transport it over the network. The migration itself might be delayed but from the perspective of the user the application remains always available.

If it is necessary to migrate very fast after the external trigger then a **connectionless network** is more opportune since it will not need an extra setup time before the delta can be transmitted.

5.7.2 Application

We limited ourselves to a very small application, the process to compute the factorial of a number. Since the environment needed to provide a high degree of computational reflection we where restricted to a very specialized programming environment: Borg.

We were not able to exploit this theme in more current environments as Smalltalk or Java.

5.7.3 Techniques

In this chapter we exploited parallelism between the evaluation of the application and the proactively migrating of a copy of that application.

The technique migrates a running application so we also need to apply the technique of strong mobility.

To be able to implement the proposed technique we need access to the computational state of a running application which is not a trivial task in current classic programming environments since they do not provide the level of computational reflection needed.

This technique might be useful when we know in advance when we are going to migrate. In ambient intelligent environments by example, where hosts are moving to each other, one may foresee that an application will migrate to a host that comes physically in the neighborhood. If we manage to send the bulk of the application in advance to the receiving host, we can, when the real time to migrate has come, obtain a high-speed migration of our running application.

At the receiver, the computational state need to be brought up to date by applying this delta to the previous received computational state before evaluation is continued. This kind of action will also require a high level of computational reflection.

We demonstrated the feasibility of the technique by migrating a small application in order to measure the difference in migration time. In our setup the technique proves to be useful, but the determination of a possible more universal nature of the technique or the demarcation of the domain in which the technique proves useful is left for future work.

6 Conclusion

6.1 Wrap-up

Ambient intelligence that builds on three recent key technologies: *Ubiquitous Computing, Ubiquitous Communication and Intelligent User Interfaces*, poses new challenges to build the underlying software in order to support cooperating systems. These systems will feature dynamic context and unpredictable connection times between a diversity of devices with their own autonomic characteristics.

The emerging technique of mobile code is a new promising way to set up communication mechanisms between different parties but there is still much research needed to develop techniques to support and optimize these communication mechanisms.

This thesis explores possibilities to hide network latency that can become very high if a block of code cannot be migrated as a whole in an environment where the width of the migration timeframes is unpredictable.

A possible solution is to break up the block of code in smaller parts and send them one by one to the receiver. This will increase the possibility that they will fit in the temporal timeframe. Precaution should be taken to send the most important parts first, in a format that makes this partial block of code immediately usable (ready for evaluation) at the receiving host.

Since connections between hosts in these new environments are more volatile than in static networks there is also the need for mechanisms that allows the code to continue its evaluation during the progressive migration so that the application remains available for users or other applications at all time.

In order to break open this new, complex and difficult research domain we explored three different themes to take advantage of the implicit parallelism found in computer networks.

It was our goal to provide a proof of concept of the different scenarios developed under these themes without pursuing completeness or universality.

6.2 Results

For the three themes we explored, pre-fetching of permuted code, component streams and proactive migration we delivered a proof of concept and we showed that for the experiments we performed under these themes, progressive mobility proved to be useful.

At the end of each chapter, we presented a table with some indications about the performance obtained under these themes. Table 22 summaries these indications from the three tables. One should take caution however by interpreting this table because a vertical comparison is almost meaningless since each theme explored different experiments under very different conditions and restrictions.

Again, we mention that performance of an application is most commonly measured by *overall program evaluation time* and network performance is most commonly measured in *network latency* but in a mobile environment performance is also measured by *application availability, invocation latency*, and *user interface latency*.

Overall program evaluation time is the time between the invocation of an application and the end of the evaluation of the last instruction.

Application availability is the inverse of the time an application “freezes” during migration. Especially in control engineering environments this may be a critical property.

Network latency is the time the application needs to travel over the network.

Invocation latency is the time from application invocation to when evaluation of the program actually begins.

User interface latency is the time a user has to wait between his demand and a user interface reaction of the system. From the viewpoint of the user this is the most crucial latency.

Table 22: Summary of the performance indications

	Overall program evaluation time	Application availability	Network latency	Invocation latency	User interface latency
Pre-fetching of permuted code	++	+	+++	++	+++
Component streams	-	+++	+++	+	++
Proactive migration	+	++	++	++	+

Table 22 summaries the performance indications in each presented theme. Again, note that a vertical comparison is not meaningful.

Progressive anticipative mobility using pre-fetching of permuted code

The results of our proof of concept excel in hiding user interface latency. Exploiting parallelism between loading and evaluation in our experiment reduced user interface latency considerably (21% of the original time on average in the three applications tested). The overall program evaluation time decreases since not all the code has to be transported and compiled. The overall program evaluation time could also be significantly reduced (79% of the original time on average in three applications tested).

Progressive mobility using component streams

The results of these experiments excels in application availability and the hiding of network latency but in general we expect the overall program evaluation time to increase since the application becomes distributed while streaming. In some cases however we might exploit the temporary parallelism to compensate for the distribution and for small applications in some test environment as in ours we might even see a decrease in evaluation time.

With progressive mobility using component streams in our setup the running code is never halted and therefore will keep its ability to react to incoming events.

We discussed the relation between migration time and idle time of the components that constitute the application and described the necessary conditions for removing network latency completely. We compared different migration strategies for progressive mobility

using component streams, and showed with our experiment that it is possible to migrate a running application autonomously and under the control of a supervisor component as if there where no network latency at all.

In our experimental setup the migrating application even runs faster during migration than when it runs stationary. We where also able to start the visual presentation part of the application before the complete application was migrated thereby gaining the same advantages for user interface invocation as the progressive anticipative mobility using pre-fetching of permuted code technique. We also showed that is possible to take advantage of parallelism between the evaluation of components on the sender and the receiver. Based on our experiments we provided some design guidelines for developing new mobile streaming applications. The determination of the universal nature of the technique or the demarcation of the domain in which the technique proves useful is left for future work.

Progressive, anticipative mobility using proactive migration

In the small experiment we conducted, we demonstrated that the technique may excel in reducing the network latency especially for the transmission of mobile applications over a slow network. The technique is based on the observation that we can calculate the delta between two computational states and that this delta is always smaller than the original computational state.

We conclude that this technique might be useful when we know in advance when we are going to migrate. In ambient intelligent environments by example, where hosts are moving to each other, one may foresee that an application will migrate to a host that comes physically in the neighborhood. If we manage to send the bulk of the application in advance to the receiving host we can, when the real time to migrate has come, obtain a high-speed migration of our running application in a fraction of the time needed for normal migration. Our experiment showed that the reduction in size of the data to transport and thus, possible also the migration time is 2.01% of the original stream size or a compression ratio of: 97.99%.

Whatever the behavior of our program, the delta of two snapshots will never be bigger than the original and therefore we will always obtain a gain in time even when we don't know in advance when we'll migrate. Therefore, we could send the computational state every few instructions to a potential receiving host. Or we could just transmit the program code itself to all potential receiving hosts at the beginning of the evaluation. The size of the code is constant and therefore should only be sent once. The determination of the universal nature of the techniques or the demarcation of the domain in which the technique proves useful is left for future work.

6.3 Discussion

We explored in the three themes some existential examples in order to show that it is possible to hide network latency by partitioning mobile code and exploit parallelism. As for now, it is still unclear how the implementation environment of ambient intelligence entities will look like but since such a system needs to have many dynamic properties, mobile code will be one of the key technologies to bring ambient environments to live. The explored themes suggest different ways to harness the implicit parallelism found in even the most simple computer networks but massively available in future ambients.

Validity of the result under current research restrictions

In order to obtain our results we applied several research restrictions (see section 1.5 page 23). Most of our experiments are conducted in fixed TCP/IP networks, but an ambient intelligence environment, our environment under research, will rely on unpredictable networks.

However, this is not contradictory. The support for these unpredictable networks needs to be offered by the distributed operating systems. In order to manage the ambient intelligent environment these operating systems will need to put systems in place to cope with these unpredictable networks and one of the tools we offer from this research path is the partition of mobile code in several parts to enhance the chance to fit in an unpredictable time frame.

We did not run experiments in connection-based networks but we argued in the different chapters that even with the extra setup time in a network the techniques might remain useful.

We feel that security aspects are orthogonal on all practical implementations of computer systems and although partitioning code may introduce extra vulnerability in the system more and more techniques to tackle these kind of security problems arise at the horizon or are already introduced and standardized.

If there is a difference in the processing power of the hosts there will arise extra opportunities to exploit this by running the most important parts of the code on the most powerful processors but the basic scenario's we introduced in the three themes will remain valid since we never assumed a difference or equality of the hosts processing power.

We only applied a push strategy since we assumed that the know-how and know-when of the migration of partitioned code is located in the sending host. However, this does not exclude the possibility of successful combinations with a pull-strategy or with a pure pull-strategy. The coordinating software that steers the progressive mobility may be located at the sender, the receiver or even a third party and was never a critical issue in our experiments.

As far as the technical restrictions of the programming environments are concerned we where always able to find a workaround for our proof of concepts.

However, in an ambient intelligence environment, build in current programming environments and with current tools, we will not be able to implement all the themes we explored in this thesis since we will largely depend on the support for strong mobility, powerful communication mechanisms between autonomous components and computational reflection provided by the system.

The proposed partitioning of code in this thesis is one of the first steps to the challenging problem in order to integrate computers into an ambient intelligence environment.

6.4 Future Work

We explored different themes in order to break open a new, complex and difficult research domain. We took the first steps in the exploration of possible network latency problems in upcoming ambient intelligence environments.

Even in the limited domain we explored there is still a lot of research to do. In this chapter we propose some interesting directions to further complete the explored themes.

6.4.1 Evaluate the Themes with other Criteria

During the exploration of the themes we focused mainly on the performance of the migration process. There are however other interesting criteria that justify extra research. We mention:

- Memory footprint
- CPU consumption
- Connection oriented networks
- Unstable networks
- Vulnerable networks
- Development overhead
- Maintainability of partitioned code
- Different processor speeds at sender and receiver
- Pull strategy
- Progressive mobility of data
- Practical experience, a field-test
- Deployment with realistic, reusable methodologies in a professional setting
- Different programming environments
  - C++
  - Java
  - .NET

6.4.2 Other Topics Related to Pre-fetching of Permuted Code

For the progressive anticipative mobility using pre-fetching of permuted code technique we need to develop a more formal approach to decide where to cut the original code and how and where to add semaphores or other guarding systems. Just cutting a permuted file in four equal parts will not always be sufficient.

More experiments are necessary to determine the optimal number of parts, but as shown in the examples a simple heuristic of cutting the source in four pieces and trying to put the first break at the point where the first GUI is built provides already significant results.

We also need to guarantee that the resulting source code behaves exactly in the same manner as the non-permuted version. For example: in multi-threading programs each thread should be guarded separately and for a reflective application that reasons over its own source code we need to take in account this special dependency.

Since the progressive anticipative mobility using pre-fetching of permuted code methodology proves to be very generic and applicable to all systems where code needs to be moved before it is evaluated, cache optimization may become a target of the proposed technique. Cache loading could be triggered based on high level abstractions of the original source code.

We plan also to use the technique for languages that internally represent their code as an abstract parse tree. One of the interesting properties of Pico [D'Hondt 2003] is that the program is internally represented as an abstract parse tree. Evaluation of the program is then evaluated by evaluating the parse tree, i.e., the tree nodes are traversed in the order imposed by the original Pico program. The traversal order is called the evaluation sequence. By tracing the evaluation sequence of the nodes it would be possible to send the program node by node to an other Pico evaluator which can start the evaluation of the nodes as they arrive, producing the same advantages as shown in our Smalltalk experiments.

6.4.3 Pre-fetching of Permuted Code with Multi Node Hopping

An application can only migrate directly to another host if the physical topology of the network allows this. In other cases applications may need to pass trough other hosts before they can reach their final destination. If an application is sent in a pre-fetched way the total transport time can be reduced significantly. Figure 60 shows the sequence diagram of an pre-

fetched loaded application that passes trough host 2 before it reaches its final destination host 3. Host 2 does not have to load the complete application before it is able to start forwarding it to the next host. The total invocation time will become approximately the load time of the first part multiplied by the number of host it visits.

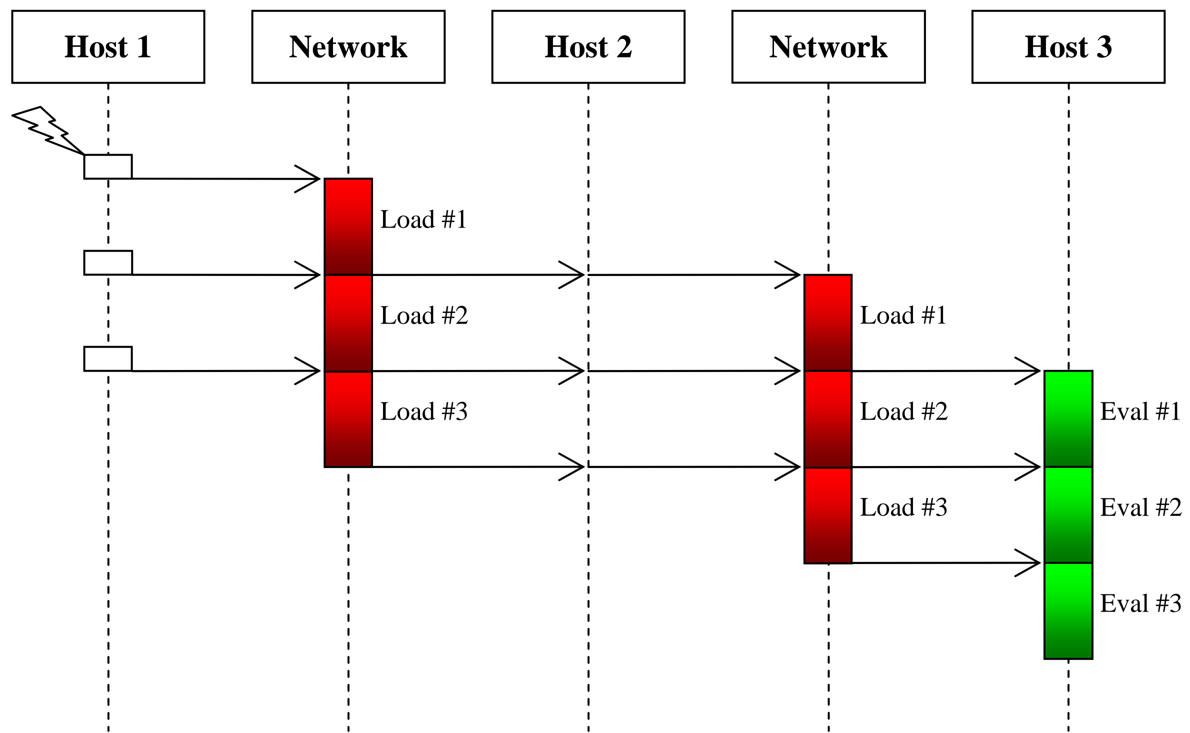

Figure 60: Multi node hopping

At a lower level in the network layers, the network layer is concerned with getting packets from source to the destination. Getting to the destination may require many hops at intermediate routers along the way. For communication in the internet the transport layer takes datastreams and breaks them up into datagrams. In theory, datagrams can be up to 64 KiB each but in practice they are usually not more than 1500 bytes so they fit in one Ethernet frame [Tanenbaum 2003]. So if the size of the datagrams is smaller than the size of the application parts a similar effect is obtained at a lower level. The fact that the first part of the application can be evaluated immediately after arrival remains due the pre-fetching at the transport layer.

Some applications are evaluated on different hosts. In this case the speeding up effect of pre-fetching the application will increase even more.

One of the first advertised applications of mobile agents lies within the field of E-commerce. Agent technology would help the user when purchasing certain goods [Chavez, 1997]. Consider a pricing agent, which helps the user to obtain the lowest possible price for a given good. Let us imagine that, as an example, the user wishes to buy an mp3 player. The agent, which is located on the user's machine, will request the specifications the player should have, e.g. the number of songs it can contain and a maximum price. Once the specifications are gathered, the agent will migrate itself towards different known vendors of such players, and at each vendors' location request the prices of mp3 players matching the specifications. When all vendors have been visited, the agent will return to the user, and at this point it will present the information it has gathered.

Figure 61 shows a sequence diagram of such a multi node hopping and evaluation application. As the figure indicates, the evaluation of the application on host 3 can start immediately after the load time of the first part of the application. In this case the loading of part #3 of the application to host 1 runs in parallel with the evaluation of part #2 on host 2, the loading of part #2 to host3 and the evaluation of part#1 at host3.

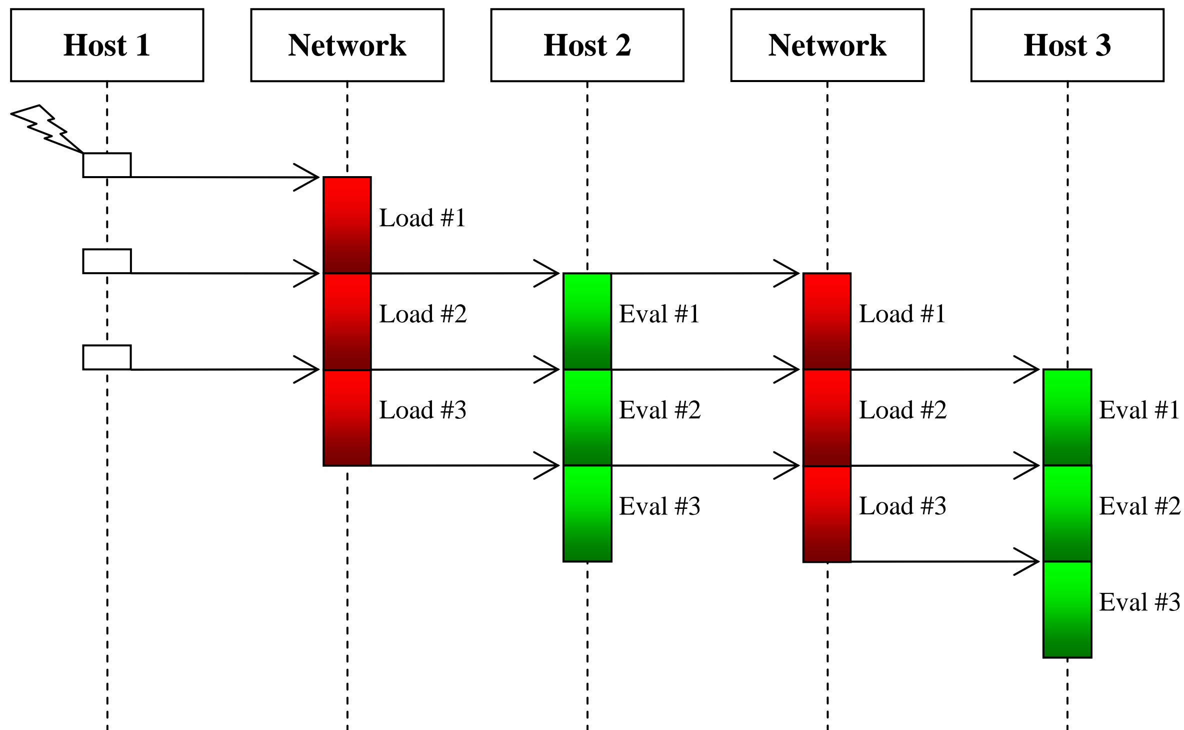

Figure 61: Multi node hopping and evaluation

6.4.4 Architectural Transformations to make Applications Streamable

Not all existing applications are suited for applying the technique of **progressive mobility using component streams** but we believe that architectural transformations can be carried out to make the proposed technique applicable. The proposed transformations should be investigated in and implemented in a transparent way.

There currently exists a trend to make a software system comply with many important non-functional requirements, such as reusability, extensibility and adaptability, enabling the developers to reuse major parts of it. This leads to systems in which a lot of attention is paid to the global architecture. How the different classes in a system are combined and the specified ways in which their objects interact becomes very important in order to be able to easily reuse or extend the system. Proof thereof is the tremendous success of programming conventions such as design patterns [Gamma et al. 1995]

The only way to increase the idle time per component is optimizing the system by transforming the architecture but this should not interfere with the architecture as defined and viewed by the designer but instead this transformations should occur during a optimization step of the compiler. [Tourwé and De Meuter 2001]

Current compilers are not able to optimize significantly highly flexible systems because they cannot automatically infer the intentions of the developer. A compiler does not know, for example, why a specific abstraction is introduced, so it cannot eliminate it or introduce other

abstractions to produce better code. Therefore, these intentions should be made explicit as in [Tourwé and De Meuter 2001] where an annotation language is provided in which these intentions can be expressed. In order for the compiler to be able to use this information in a useful way, it should incorporate some knowledge on how to optimize a certain intention. Again, this knowledge should be provided by the developer and can be expressed in the transformation language.

A lot of user intervention is required for the optimization of a system. We believe this is unavoidable however, as systems tend to get more complex and because there are limits to the amount of information that can be deduced automatically by dataflow analysis techniques [Zima and Chapman 1990].

Optimizing architectural transformations for object-oriented languages will resemble refactorings [Fowler et al. 1999]. Refactoring is a technique to restructure code in a disciplined way. For a long time it was a piece of programmer lore, done with varying degrees of discipline by experienced developers, but not passed on in a coherent way. [Fowler et al. 1999]

In this context, refactorings should be interpreted as optimizations. The most obvious reason to optimize an application for streaming is converting it from a coarse-grained componentization to a fine-grained componentization in order to allow components to migrate as independently as possible. The ultimate goal of a refactoring process is to restyle the application in such a way that the model it describes maps as closely as possible to the model of the part of the real world it tries to emulate. In our case where only optimization is pursued this is not a concern. Our only goal here is to reduce the size of the components.

6.4.5 Progressive Mobility using Proactive Migration and evaluation

This technique is proposed as a new progressive migration scheme that also sends the code anticipative to the remote host but, instead of waiting for the computational state update, the incomplete application is proactively launched at the receiving host the same moment the migration is triggered and the new computational state is fetched at run-time. This could potentially allow applications to migrate at zero time.

Proactive migration avoids the temporary distribution of *progressive mobility using component streams* and may allow the migrating application to run almost continuously at full speed. The application might be only stalled during the period the computational difference is calculated and transferred. Although this time might be only a few percent of the normal migration time there is still some time the application is not available for other processes or users.

Proactive migration and evaluation can go one final step further by eliminating the waiting time completely by not only migrating anticipatory the application but also by starting its evaluation up proactively. It is a generalization of the special technique dealt with in previous section (5.6.4). Proactive migration and evaluation is a technique that needs certain precautions to enable an immediate startup of the application in all possible cases. To demonstrate the feasibility of the technique we built a simple prototype in Smalltalk.

6.4.5.1 Proposed Technique

The technique of proactive migration and evaluation also applies a progressive migration scheme. The running application is split in two components: a snapshot of the complete application and the delta of the computational states that will be fetched at run time. Basically the technique is a five step process:

1. Take a **snapshot** of the running application, i.e. take a copy of the code and its computational state, on the sending host.
2. **Copy** the snapshot to the receiving host **while the original application continues to run.**
3. Once the copy has arrived at the receiving host **start** the application at the receiving host.
4. **Halt** the original application and define the changes, called **the delta,** which emerged during the copy phase of the snapshot. This delta contains the changes in the computational state.
5. Migrate and **apply this delta** to the, already running application, in the same spirit as code pre-fetching, i.e. fetch the part of the state first that is needed first.

Since the application at the receiving host is always started before the original sending application is halted the application could be always available and the migration will happen in zero time. A sequence diagram that illustrates this technique is shown in Figure 62.

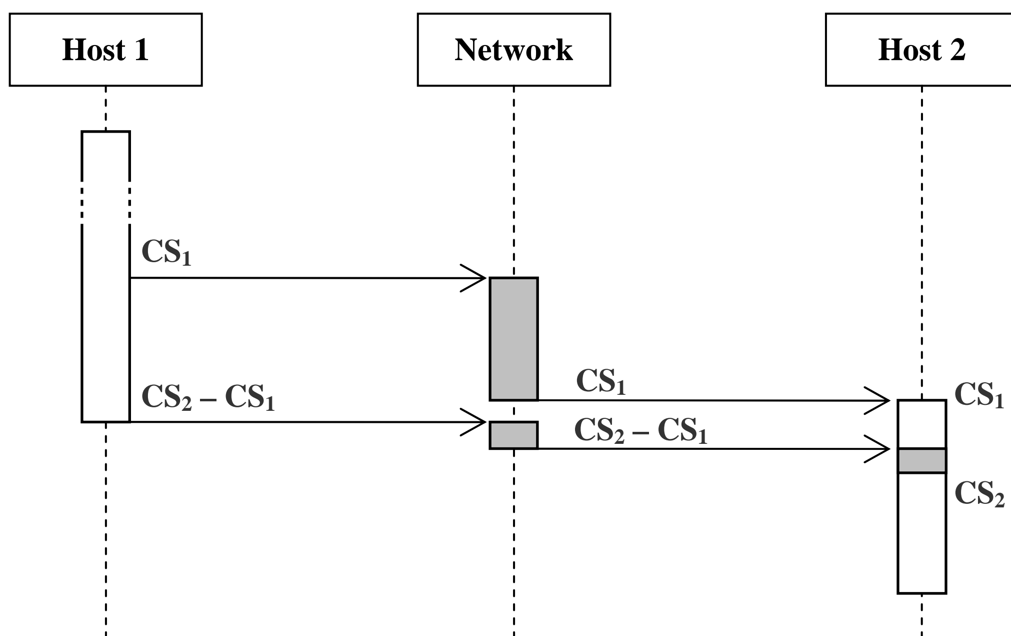

Figure 62: Proactive migration and evaluation

6.4.5.2 Assumptions and Restrictions

Applying the computational state to a running application is far from trivial. The most challenging aspect of it is that by applying a new computational state, return addresses of running functions or methods can change and in the case of recursive algorithms, the complete run-stack with return addresses and intermediate results can become completely different.

We may however put some restrictions on the content of the computational state to make it more manageable.

Basically a computational state contains the current evaluation position (a high level equivalent of the *program counter,* a register in the hardware processor), the values of application variables and implicit return addresses of subroutines, possible in a recursive call chain.

The current evaluation position is used when the application is launched at the receiving host and does not need adaptation later on. This leaves us with the values of application variables and return addresses of subroutines.

If we could get rid off the return addresses and restrict the computational state to the set of values of the application variables this would make it much easier to adapt it at run-time.

There might be a way to get rid off implicit return addresses in a computational state. You may eliminate the administration of return addresses by just not going anywhere.

Implicit stack operations could be avoided by adapting the algorithms so that the data and return address stack is made explicit in the application. In practice this implies that in the application an explicit stack structure will be declared and that the application itself becomes responsible for maintaining its state (loop counters and intermediate results) on this stack.

Calls to subroutines could be avoided by applying code inlining to the complete application. Code inlining is an optimization technique to speedup code just by getting rid of the return address management. Inlined code tends to grow bigger which may be not ideal for mobile applications. In our setting however, where static code is migrated in advance this may not be such a disadvantage. The equivalent of code inlining in object oriented programming languages is object inlining [Andrew and Chien 2000].

If both techniques, avoiding of implicit stack operations and code inlining, are applied, our computational state is reduced to the set of instance variables of the main application.

6.4.5.3 Handling Semaphores

Even when the computational state of an application is reduced to its instance variables they should be handled with care. When the application is started proactively before its state is adapted one should take care not to rely on values that are possibly changed at the sender's version of the program. The safest way is to guard all the variables by semaphores, in the same spirit as the code parts were guarded in the pre-fetched code technique, and suspend evaluation when one of them is accessed.

If a variable is accessed for a write instruction then it doesn't matter if the original is changed or not since the result is destroyed anyhow. This may allow us to relax the guarding policy somewhat so that only read instructions (*get* methods) need to be guarded.

If a variable is updated by the sending host then the semaphore can be inactivated at the same time. The semaphore can also be physical removed now or this task can be delegated to dedicated garbage collecting agents.

When the migration is triggered at the sending host and before the computational state changes are transferred a kind of change summary could be send in advance as a real avant-première. This summary will allow the receiving host to optimize its semaphore settings by guarding only read instructions on variables that where not changed at the sending host since the proactive migration.

Proactive migration and evaluation can only be deployed successfully if the sequence of adding the new computational state is chosen in such a way that the running application is not delayed. Developing methods to permute the presentation of the computational state so that it fits seamlessly in the running receiver may build further on the profiling techniques applied for progressive anticipative mobility using pre-fetching of permuted code. The development of algorithms to automatically inline objects and to avoid implicit stack operations will allow a simplification of the presentation of the computational state. This will in turn facilitate computational state adaptation at runtime. Also these proposed transformations should be

implemented in a transparent way so that they do not interfere with the architecture as defined and viewed by the designer but instead as the architectural transformations proposed for progressive mobility using component streams, these transformations should occur during an optimization step of the compiler.

The result could be that, providing that the delta is applied in the ideal sequence, the migration can take place in no time. The application remains available at all time and the perceived network latency is virtually reduced to zero.

6.4.6 Aspects

The dynamic behavior of the different proposed techniques is clearly identifiable but scattered over many places in the software, cross-cutting the different software components. Aspect Oriented Software Design is an upcoming software engineering technique that promises the possibility to describe aspects of the proposed techniques in an insulated modularized component. Aspect-oriented programming (AOP) is based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying an mechanisms in the underlying AOP environment to weave or compose them together into a coherent program [Elrad et al. 2001].

The introducing of distribution and migration generates cross-cutting concerns to the system as there are: security, encryption and authentication of the code, buffering, proxy and routing concerns, distributed transaction issues and the management of different possible modes of transparency:

- Location transparent

  User can not tell where resources are located

- Migration transparent

  Resources can move at will without changing their names

- Replication transparent

  User cannot tell how many copies exist

- Concurrency transparent

  Multiple users can share resources automatically

- Parallelism transparent

  Activities can happen in parallel without users knowing

In the different progressive migration techniques presented in this dissertation we distinguish two kinds of source code where an aspect oriented approach might be appropriate, the source code of the application to migrate (basic code) and the source code of the supervising system that guides the migration (supervising code). For each progressive migration technique we identify some crosscutting concerns that might be aggregated in aspects.

- Progressive Anticipative Mobility using Pre-fetching of Permuted Code
  - Basic code

    Instrumentation of the source code with extra code that logs the time of invocation of each method.

    Instrumentation of the source code with extra code in order to delimit the code needed to build the GUI.

Synchronization code (semaphores).

Code for removal of synchronization code after first use.

- o Supervising code

  Determines the number of files to create and therefore the join points in the basic code, those places where aspect code interacts with the rest of the system.

- Progressive Mobility using Component Streams
  - o Basic code

    In a *self triggered* strategy each component need to contain the same block of code to decide when and how to migrate itself.
  - o Supervising code

    Determines the size of the components and therefore the join points in the basic code.
- Progressive Anticipative Mobility using Proactive Migration
  - o Basic code

    Instrumentation of the source code with extra code that logs the variations in the computational state to be able to determine possible proactive migration times and to detect possible large delta's in the computational state.
  - o Supervising code

    Addition of dynamically computational state monitoring code to the basic code during its evaluation.

6.4.7 New Research Projects

One of the most satisfying results of this research is that it will be embedded as a topic in new upcoming research projects that plan to deepen certain directions and/or will embed the techniques in a bigger framework.

A new research project **@Media** (Advanced Media) is approved as a successor of the MPEG-project and is aimed at developing prototypes and basic research. This new e-VRT research project, in close cooperation with our national radio and television broadcast company has started end 2003.

This project is part of the effort to develop a Content Management System to manage new and existing content material (images, sound, graphic content, games, interactive scenarios and is situated around mobile code and MPEG-4 [Puri and Eleftheriadis 1998] environments. This setting will give us the real live test environment to validate our approach further on different platforms and will allow us to get more detailed results. The project has not only the ambition to manage and support multi channel publications of data but also behavior (i.e. code). If users should be able to interact with certain scenes an application to support this interaction need to be available at the users platform as soon as possible. This is where our proposed techniques can possible play an important role.

Experiments with progressive anticipative mobility using pre-fetching of permuted code, progressive mobility using component streams, proactive migration and proactive migration and evaluation will be performed to migrate the code needed for interactive television to the clients.

Also a strategic basic research project **CoDAMoS** (Context-Driven Adaptation of Mobile Services) is proposed and is aimed at solving a set of key challenges in the area of ambient intelligence where personal devices will form an extension of each user's environment, running mobile services adapted to the user and his context.

One of the work packages includes: **Progressive mobility.** The objective is to establish a framework for progressive mobility, i.e. a feature of mobile code whereby execution and migration are interleaved in such a way that network latency is minimized. Our results will be taken as a starting point.

In a first stage, a high-level virtual machine developed in another task will be used to explore the various concerns of progressive mobility: partial ordering of code fragments using symbolic interpretation (how is migration sequenced?), pushing or pulling code fragments (who takes the initiative for migration?), evaluation overlap (destination evaluation start before source evaluation stop), etc. In a second stage, a pragmatic subset of the results obtained with the high-level virtual machine will be applied to a Java context, using aspect technology.

Bibliography

Amdahl 1967 Amdahl, G.M. *Validity of the single-processor approach to achieving large scale computing capabilities.* AFIPS Conference Proceedings vol. 30 (Atlantic City, N.J., Apr. 18–20). AFIPS Press, Reston, Va. , pp. 483–485. 1967

Andrew and Chien 2000 J.Dolby, Andrew, A. Chien. *An Automatic Object Inlining Optimization and its Evaluation* ACM SIG\-PLAN Notices, 2000

Arnold et al. 1999 K. Arnold, B. O'Sullivan, R.W. Scheiffer and J. Waldo, A. Wollrath. *The Jini Specification*. Addison-Wesley, 1999

Attali et. Al 2000 Isabelle Attali, Denis Caromel and Romain Guider, in *FMOODS 2000, A Step Toward Automatic Distribution of Java Programs* Stanford University, Kluwer Academic Publishers, pp. 141-161, 2000,

Barbacci 95 Barbacci Mario, Klein Mark H., Longstaff Thomas H. & Weinstock, Charles B. *Quality Attributes* (CMU/SEI-95-TR-021). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1995

Berners-Lee 1996 T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol HTTP/1.0*. RFC 1945. Network Working Group, May 1996

Caromel et al. 1998 D. Caromel, W. Klauser, J. Vayssiere, *Towards Seamless Computing and Metacomputing in Java* Concurrency Practice and Experience, pp. 1043--1061 Editor Geoffrey C. Fox, Published by Wiley & Sons, Ltd. 1998

Carzaniga et al. 1997 Antonio Carzaniga, Gian Pietro Picco, Giovanni Vigna. *Designing Distributed Applications with Mobile Code Paradigms.* Proceedings of the 19th International Conference on Software Engineering, 1997

Chavez 1997 A. Chavez, D. Dreilinger, R. Guttman, and P. Maes. *A real-life experiment in creating an agent marketplace*, Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology, April 1997

Chen 2002 Guanling Chen and David Kotz. *Solar: An Open Platform for Context-Aware Mobile Applications*. In Short Paper Proceedings of the First International Conference on Pervasive Computing (Pervasive 2002) (PDF), August, 2002

Chilimbi et al. 1992 T. Chilimbi, B. Davidson, and J. Larus. *Cache-conscious structure/class field reorganization techniques for c and Java*. Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation, May 1999.

Cincom 2003 *Welcome to Cincom Smalltalk VisualWorks®* Non-Commercial Pre-Release 7.1 Cincom Systems, Inc of March 19, 2003..

Devalez 2003 Christian Devalez. *Application streaming in Java*. Licentiaat thesis, faculteit wetenschappen, departement informatica en toegepaste informatica, Vrije Universiteit Brussel, ftp://prog.vub.ac.be/dissertation/2003/Devalez2003.pdf, Brussel 2003

Devaney 1992 Robert L. Devaney. *Chaos, Fractals & Dynamica: Computer-experimenten in de wiskunde*. Addison-Wesley Nederland, p. 105, 1992

D'Hondt 2003 T. D'Hondt. *Pico: programming language.* http://pico.vub.ac.be, June 2003

Doherty and Kelisky 1979 W. J. Doherty and R. P. Kelisky. *Managing VM/CMS systems for user effectiveness*. IBM Systems Journal, pages 143–163, 1979

Dürst 1997 M. J. Dürst. *The progressive transmission disadvantage*. IEEE Transactions on Information Theory, Vol. 43, No. 1, , pp. 347-350. Jan. 1997

Edward 2001 Edward B. Allen, Taghi M. Khoshgoftaar, Ye Chen Edward. *Measuring Coupling and Cohesion of Software Modules: An Information-Theory Approach.* Seventh International Software Metrics Symposium pp. 124, April 2001

Elrad et al. 2001 Tzilla Elrad, Robert E. Filman, Atef Bader. *Aspect-oriented programming: Introduction.* Communications of the ACM, Volume 44 Issue 10 p. 29-32 October 2001

Ernst et al. 1997 J. Ernst , W. Evans , C. W. Fraser , T. A. Proebsting , S. Lucco. *Code Compression*. Proc. ACM SIGPLAN conf. on Programming language design and implementation. Volume 32 Issue 5, May 1997

Evenepoel 2003 Karl Evenpoel *Progressive mobility in Smalltalk using Opentalk*. Licentiaat thesis, faculteit wetenschappen, departement informatica en toegepaste informatica, Vrije Universiteit Brussel, ftp://prog.vub.ac.be/dissertation/2003/Evenepoel2003.pdf, Brussel 2003

Fisher 1992 Joseph A. Fisher, Stefan M. Freudenberger. *Predicting conditional branch directions from previous runs of a program*. ACM SIGPLAN Notices , Proceedings of the fifth international conference on Architectural support for programming languages and operating systems, Volume 27 Issue 9 September1992

Fowler et al. 1999 Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley publishing company isbn: 0201485672; 1st edition, June 28, 1999

Franz and Kistler 1997 M. Franz and T. Kistler. *Slim Binaries*. Comm. ACM Volume 40 Issue 12, December 1997

Fuggetta et al. 1998 Alfonso Fuggetta, Gian Pietro Picco and Giovanni Vigna. *Understanding Code Mobility*. IEEE Transactions of Software Engineering, volume 24, 1998

Gamma et al. 1995 Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley professional Computing Series, 1995

Ghim and Chorng 2001 O. Ghim-Hwee, C. Chorng-Meng C. Yi. *A simple partitioning approach to fractal image compression.* Proc. 16th ACM SAC2001 sym. Applied computing, Las Vegas, Nevada, US ACM Press Pages: 301 – 305, 2001

Goldberg and Robson 1989 Adele Goldberg and David Robson. *Smalltalk-80 the language* Addison-Wesley publishing company isbn: 0-201-13688-0, 1989

Gruia-Catalin 2000 Gruia-Catalin Roman, Gian Pietro Picco & Amy Murphy. *The Future of Software Engineering.* Anthony Finkelstein (Ed.) ACM Press, 2000

Grune 2000 D. Grune, H. Bal, C. Jacobs, K. Langendoen. *Modern Compiler Design Worldwide Series in Computer Science.* John Wiley & Sons ltd London, 2000

Hauswirth 1999 Manfred Hauswirth. *Internet-Scale Push Systems for Information Distribution Architecture, Components, and Communication*. PhD dissertation at Technischen Universität Wien, August 1999

Hoare 1985 C.A.R. Hoare. *Communicating sequential Processes.* Prentice Hall International Series in Computer Science, 1985

IEC 2000 IEC 60027-2, Second edition, *Letter symbols to be used in electrical technology* - Part 2: Telecommunications and electronics, November 2000

IEEE 1997 *A Lesson in Megabytes*. IEEE Standards Bearer, page 5. Portions copyright © 1997 by the Institute of Electrical and Electronics Engineers Inc. January 1997

Intel 2002 *White paper: Hyper-Threading Technology on the Intel® XeonTM Processor Family for Servers.* Intel corporation, 2003

ISTAG 2001 *Scenarios for Ambient Intelligence in 2010.* ISTAG Final Report, EC, Feb 2001

Jason and Patterson 1995 R. Jason, C. Patterson. *Accurate Static Branch Prediction by Value Range Propagation.* Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation, pages 67-78, June 1995

Johnson 1998 Chris Johnson. *The Ten Golden Rules for Providing Video Over the Web or 0% of 2.4M (at 270k/sec, 340 sec remaining)*. Human Factors and Web Development, section 16: pages 207–221. Lawrence Erlbaum Associates, Publishers, Mahwah, New Jersey, 1998

Jones 1996 Neil D. Jones. *An introduction to partial evaluation*. ACM Computing Surveys (CSUR) Volume 28 Issue 3 ,September 1996

Krintz et al. 1998 Chandra Krintz, B. Calder, H. B. Lee, B. G. Zorn. *Overlapping Execution with Transfer Using Non-Strict Execution for Mobile Programs*. Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems, San Jose, California U.S., October, 1998

Krintz et al. 1999 Chandra Krintz, B. Calder and U. Hölzle. *Reducing Transfer Delay Using Class File Splitting and Pre-fetching*. Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications, November, 1999

Krintz 2001 Chandra Krintz. *Reducing Load Delay to Improve Performance of Internet-Computing Programs* PhD dissertation, UCSD Technical Report CS2001-0672, May, 2001

Ladin et al. 1992 Rivka Ladin, Barbara Liskov, Liuba Shrira and Sanjay Ghemawat. *Providing High Availability Using Lazy Replication.* MIT Laboratory for Computer Science, ACM Transactions on Computer Systems, Vol 10, No. 4, Pages 360–391, November 1992

Lange et al. 1998 Lange, D.B. & Oshima M. *Programming and Deploying Java Mobile Agents with Aglets.* Reading, Addison-Wesley, Massachusetts, 1998

Lee 1997 Han Bok Lee. *BIT: Bytecode instrumenting tool*. Master's thesis, University of Colorado, Boulder, Department of Computer Science, University of Colorado, Boulder, CO, June 1997

Le Gall 1991 D. Le Gall. *MPEG: a video compression standard for multimedia applications.* Communications of the ACM Volume 34 Issue, 4 April 1991

Lindwer et al. 2003 Menno Lindwer, Diana Marculescu, Twan Basten, Rainer Zimmermann, Radu Marculescu, Stefan Jung, Eugenio Cantatore, *Ambient Intelligence Visions and Achievements: Linking Abstract Ideas to Real-World Concepts*. Design, Automation and Test in Europe Conference and Exhibition (DATE'03) Munich, Germany, March 2003

Maes 1987 P. Maes. *Computational Reflection*. PhD thesis, Vrije Universiteit Brussel, 1987

Milojicic et al. 1999 Milojicic Dejan, Douglis Fred, Paindaveine,Yves, Wheeler Richard, Zhou Songnian. *Process Migration.* HP Labs Technical Reports HPL-1999-21 990217 External, 1999

Moore 1965 G. Moore. *Cramming more components onto integrated circuits.* Electronics, Vol. 38(8), pp. 114-117, April 19, 1965

Nejmeddine 2002 Nejmeddine Tagoug. *Object-Oriented System Decomposition Quality.* 7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02), pp. 230, October 2002

Nierstraz 1995 Oscar Nierstrasz and Theo Dirk Meijler. *Research di-rections in software composition*. ACM Computing Surveys, 27(2):262-264, June 1995.

O'Hare et al. 2004 G. M. P. O'Hare, M. J. O'Grady, S. Keegan, D. O'Kane, R. Tynan & D. Marsh, *Intelligent Agile Agents: Active Enablers for Ambient Intelligence*, Ambient Intelligence for Scientific Discovery (AISD) SIGCHI Workshop, Vienna April 25, 2004

Picco 1998 G. P. Picco, *μCode: A Lightweight and Flexible Mobile Code Toolkit.* Mobile Agents, Proceedings of the 2ndInternational Workshop on Mobile Agents 98 (MA'98), Stuttgart (Germany), K. Rothermel and F. Hohl eds., Springer, Lecture Notes on Computer Science vol. 1477, pp. 160-171, September 1998

Picco 2001 G. Pietro Picco. *Mobile Agents*. 5th International conference, MA 2001 Atlanta Preface proceedings, 2001

Plezbert and Cytron 1997 M. P. Plezbert , Ron K. Cytron. *Does "just in time" = "better late than never"?* Proc.24th ACM SIGPLAN-SIGACT sym. on Principles of programming languages, p.120-131, Paris, France, January 15-17, 1997

Puri and Eleftheriadis 1998 A. Puri, A. Eleftheriadis. *MPEG-4: An object-based multimedia coding standard supporting mobile applications.* Mobile Networks and Applications 3 5–32, 1998

Sazeides 1998 Yiannakis Sazeides and James E. Smith. *Modeling Program Predictability.* Colloquium Series Iowa State University Spring 1998

Shaw-Kung Jong 2000 Shaw-Kung Jong, Belka Kraimeche. *QoS Considerations on the Third Generation (3G) Wireless Systems.* AIWORC'00 Academia/Industry Working Conference on Research Challenges pp. 249, April 2000

Siegel 1996 D. Siegel, *Creating killer web sites.* Indianapolis: Hayden Books, 1996

Sirer et al. 1999 E.Sirer A.Gregory and B.Bershad *A practical approach for improving startup latency in Java applications*. Workshop on Compiler Support for Systems Software, 1999

Stoops et al. 2002 Luk Stoops, Tom Mens and Theo D'Hondt. *Fine-Grained Interlaced Code Loading for Mobile Systems*. 6th International Conference MA2002, LNCS 2535, pp. 78-92 Barcelona, Spain, October 2002

Stoops et al. 2003a L. Stoops, T. Mens, T. D'Hondt. *Reducing Network Latency by Application Streaming*. International Conference on Parallel and Distributed Processing Techniques and Applications, Las Vegas, Nevada, USA, June 2003

Stoops et al. 2003b L. Stoops, T. Mens, C. Devalez, T. D'Hondt. *Migration Strategies for Application streaming*. technical report 2003 ftp://prog.vub.ac.be/tech_report/2003/vub-prog-tr-03-06.pdf, 2003

Sun 2002 Sun Microsystems. *Java Remote Method Invocation Specification*. http://java.sun.com/products/jdk/rmi/, 2002

Szyperski 2003 Clemens Szyperski, *Component Technology - What, Where, and How?* Proceedings of the 25th international conference on Software engineering, May 2003

Tanenbaum 2003 Andre S. Tanenbaum. *Computer Networks* Prentice Hall PTR, fourth edition, 2003

Tanter et al. 2003 Eric Tanter, Jacques Noyé, Denis Caromel, Pierre Cointe. *Partial Behavioral Reflection: Spatial and Temporal Selection of Reification.* OOPSLA'03, Anaheim, California, USA, October 2003

Tourwé and De Meuter 2001 Tom Tourwé and Wolfgang De Meuter. *Optimizing Object-Oriented Languages through Architectural Transformations.* Proc. Int. Conf. Software Maintenance, Firenze, November 2001

Ungar 1984 D.Ungar *Generation Scavenging: A non-disruptive high performance storage reclamation algorithm.* Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, April 1984

Van Belle et al. 2001 Werner Van Belle, Johan Fabry, Karsten Verelst and Theo D'Hondt. *Experiences in Mobile Computing: The CBorg Mobile Multi Agent System.* Tools Europe 2001, March 2001

van Loenen 2003 Evert J. van Loenen. On the role of Graspable Objects in the Ambient Intelligence Paradigm. (Soc 2003) smart objects conference, Grenoble 2003

Venkatesh 1991 G. A. Venkatesh. *The semantic approach to program slicing*. ACM SIGPLAN Notices , Proceedings of the conference on Programming language Design and implementation Volume 26 Issue 6, May 1991

Weiser and Brown 1996 Mark Weiser and John Seely Brown. *The Coming Age of Calm Technology* Xerox PARC October 5, 1996.

Wilson 1992 Paul R. Wilson. *Uniprocessor garbage collection techniques*. Proc of International Workshop on Memory Management in the Springer-Verlag Lecture Notes in Computer Science series, St. Malo, France, September 1992

Wirth 1995 Niklaus Wirth *A Plea for Lean Software*. ETH Zürich Computer, February 1995

Wolski 1998 R.Wolski *Dynamically forecasting network performance using the network weather service.* Cluster Computing, 1998

Zima and Chapman 1990 Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers*. Addison Wesley, 1990

Ziv and Lempel 1977 J.Ziv and A.Lempel. *A Universal Algorithm for sequential Data compression*. IEEE Transactions on Information theory Vol 23, No.3, May 1977