EPTCS 27

Proceedings of the

First International Workshop on Decentralized Coordination of Distributed Processes

Amsterdam, The Netherlands, 10th June 2010

Edited by: Tom Van Cutsem and Mark Miller

7th June 2010
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Preface

This volume contains the papers presented at the first International Workshop on Decentralized Coordination of Distributed Processes, DCDP 2010, held in Amsterdam, The Netherlands on June 10th, 2010 in conjunction with the 5th International Federated Conferences on Distributed Computing Techniques, DisCoTec 2010.

The central theme of the workshop is the decentralized coordination of distributed processes. Decentralized meaning there is no single authority in the network that everything is vulnerable to. Coordinated meaning processes need to cooperate to achieve meaningful results, potentially in the face of mutual suspicion. Distributed, meaning processes are separated by a potentially unreliable network.

The workshop opened with a Keynote by Tyler Close (Google) titled "Using the Web for decentralized coordination of distributed processes. You *can* get there from here." Afterwards authors presented two full papers and two extended abstracts.

We wish to thank all members of the Program Committee who carefully reviewed the submissions:

- Fred Spiessens, Evoluware, Belgium
- Carl Hewitt, MIT EECS (Emeritus), USA
- Ben Laurie, Google, UK
- Alan Karp, Hewlett-Packard, USA
- Peter Van Roy, Universit Catholique de Louvain, Belgium
- Dean Tribble, Microsoft, USA
- Toby Murray, University of Oxford, UK
- Tyler Close, Google, USA
- Mark Miller, Google, USA
- Tom Van Cutsem, Vrije Universiteit Brussel, Belgium

We also thank Tyler Close for willing to give a Keynote speech. We thank Marcello Bonsangue for giving us the chance to organize this workshop at DisCoTec, and to the Editorial Board of the Electronic Proceedings in Theoretical Computer Science (EPTCS) for publishing the workshop proceedings.

Tom Van Cutsem and Mark S. Miller
DCDP 2010 Organizers

Using the Web for decentralized coordination of distributed processes: you *can* get there from here

Tyler Close
Google, USA

At first blush, the Web might not seem like a good starting place for decentralized coordination of distributed processes. A typical Web application is vulnerable to multiple central authorities and so is not decentralized. Most often, coping with the travails of distribution depends upon human intervention via the browser's 'refresh' button; which doesn't bode well for headless processes. Coordination between Web applications, where it's done at all, often results in complete vulnerability between participants. Looking at the Web as a platform for decentralized coordination of distributed processes, it seems reasonable to conclude: "You can't get there from here".

Sometimes, a different perspective is all that is needed to find a way forward out of the maze. In this talk, we'll reacquaint ourselves with the Web's core technologies: the URL, HTTP and TLS. With a fresh outlook on these technologies, we'll explore how to use them for the desired effect, while still working within the existing Web infrastructure. Using simple and compatible extensions to the Web, we'll study cases where we can now coordinate the formerly intractable. The Waterken Server and an extended Web browser enable demonstration of these implementation techniques. With a different perspective on where "here" is, we'll get "there".

On Secure Workflow Decentralisation on the Internet

Petteri Kaskenpalo
AUT University
Auckland, New Zealand
School of Computing and Mathematical Sciences
Petteri.Kaskenpalo@aut.ac.nz

Decentralised workflow management systems are a new research area, where most work to-date has focused on the system's overall architecture. As little attention has been given to the security aspects in such systems, we follow a security driven approach, and consider, from the perspective of available *security building blocks*, how security can be implemented and what new opportunities are presented when empowering the decentralised environment with modern distributed security protocols.

Our research is motivated by a more general question of how to combine the positive enablers that email exchange enjoys, with the general benefits of workflow systems, and more specifically with the benefits that can be introduced in a decentralised environment. This aims to equip email users with a set of tools to manage the semantics of a message exchange, contents, participants and their roles in the exchange in an environment that provides inherent assurances of security and privacy.

This work is based on a survey of contemporary distributed security protocols, and considers how these protocols could be used in implementing a distributed workflow management system with decentralised control . We review a set of these protocols, focusing on the required message sequences in reviewing the protocols, and discuss how these security protocols provide the foundations for implementing core control-flow, data, and resource patterns in a distributed workflow environment.

1 Introduction

The research area of distributed and decentralised workflow management is still an immature field with many open problems. A few distributed and even fewer decentralised workflow management systems have been discussed in the literature. L. Guo et al. [9] provide an overview of the existing work, including those of [6, 25, 8]. As these earlier works do not discuss how security protocols could be used to address decentralised workflow related problems, we aim to provide a starting point with this article.

We were prompted to look into this area by the general lack of progress in responding to the needs of email users. While email is used to run various workflow process like activities, users are provided with little support in managing these. We believe that decentralised workflow management research will help in addressing this problem, and will return to this topic along our more generic discussion.

This work is based on a survey of contemporary distributed security protocols, and considers how these protocols could be used in implementing a distributed workflow management system with decentralised control. This article makes the following contributions. Firstly we review a set of contemporary distributed security protocols, focusing on the required message sequences in reviewing the protocols, and leaving out the cryptographic algorithms required to construct and process the message contents. This makes the protocols more accessible without expert knowledge in cryptography and mathematics, and enables us to focus on required interactions and involved participants. Secondly we discuss how these security protocols can provide the foundations for implementing secure core workflow control, data and resource patterns in a distributed environment.

The advancement of secure distributed, decentralised processing requires well-understood building blocks that can be used in modelling processes at a higher abstraction level. We approach this goal from the angle of contemporary security protocols aimed at solving generic problems in distributed collaboration, and consider how these solutions can be utilised in the area of decentralised workflow management. In addition to addressing known problems in a new environment, we also suggest that these enable new workflow functionality relating to privacy, anonymity and group based identity.

1.1 Motivation

Email is the most used communication tool in business today and its use has been extended far beyond its original purpose of simple information exchange [16]. One reason for this success is the asynchronous and distributed nature of the email message exchange, which by definition retains control over the communication, its contents, and time of interaction with the end-users.

The success of email has certainly also been its Achilles heel, as the tool is stretched to areas it was not designed for. Examples of everyday email uses now include: information management; task and time management; multi-party activity co-ordination; distributed decision making; negotiations; and voting as a discussion facilitator, to name a few. In many ways, email is being used to manage complex, collaborative process activities. This development has, of course, been further accelerated with the ability to access emails everywhere via mobile devices.

However, considering the relatively long development record of messaging tools, it is striking how its very success has led to its many problems; difficulties in dealing with message overflow, unsolicited messages, and message linking, archiving and recovery. Inefficiencies can be considerable, as the end-users do not have enough time to manage these intermingled, unstructured message flows.

Positive enablers, like flexibility and end-user-centredness of email systems, remain in distinct contrast with today's workflow management systems, which require centrally configured process rules and centrally managed process execution. They give little freedom to the users with regards self-organisation, task scheduling, and little inherent assurance of security and privacy due to their centrally managed nature. We suggest that it is possible to combine the end-user-centredness of email with the benefits of workflow systems in a decentralised email platform that would equip its users with a set of tools to manage the semantics of a message exchange, contents, participants and their roles in the exchange.

We have suggested the *Service Oriented Email* (SOE) [13] concept to address these challenges in a structured and controlled manner, and at the same time provide an organisational approach for solving email-related workflow process problems. Our research advances the notion of semantic email as introduced by McDowell et al. [17], and proposes a scalable distributed semantic email platform for automating complex multiparty update, query, resolution and process activities without exposing individuals beyond customary expectations for email privacy and control.

The key construct in the SOE concept is the ability of end-users to publish email message interfaces that define the service they provide and what they require for execution. By chaining these message interfaces, end users could be enabled to build complex distributed activities. The users should also be able to manage the organisational or social structures in which they participate by creating groups and managing group participants, and by linking a group with higher level hierarchical units. This bears a similarity with the paradigm of multiagent systems (*MAS*), where the provided tools for cooperation and coordination between agents are probably its most important contribution. However, rather than applying these principles to autonomous agents, we suggest to equip end-users with similar tools to assist them.

To achieve this goal, we require the support of distributed security protocols that provide the core security services for the messaging platform, and suitable modelling constructs for defining and controlling the message exchanges. However, we do not limit our discussion in this article only to the SOE environment, but take a general view of the decentralised workflow enactment problem.

We accept that all of the discussed protocols assume the existence of a basic public key infrastructure and a trusted source to assign keys to participants. However, this task could be handled by multiple participants, depending on their existing connections in the hierarchy, for example, in a company setting where all users could be set up by the human resources department, and the subsequent management of group memberships could be managed amongst the users. An alternative approach would be to provide a group key scheme establishment service that would facilitate the initial group key setup, similar to current Internet certificate providers, but that would not participate in the subsequent key management.

1.2 State of workflow systems

Workflow and process-driven systems are, of course, a well-studied area of computing. The work of Aalst et al. in *Workflow Patterns* [24] provides a comprehensive review of workflow building blocks, and discusses the workflow specifications from a number of different perspectives; namely the controlflow, data, resource and operational perspectives. They make a distinction between twenty different controlflow patterns. The work of Aalst et al. in [24] has been further advanced by Russell in [20] by analysing patterns supported by various workflow management systems, and by theoretical analysis of possible patterns not yet considered. His work suggests an additional 23 controlflow patterns and many additional patterns in the other categories.

While the works of Aalst et al. and Russell aim to *'identify comprehensive workflow functionality'* and to *'provide the basis for an in-depth comparison of a number of commercially available workflow management systems'* [1], they also provide an invaluable foundation for further study of workflow management systems, especially when moving from centrally controlled, non-distributed workflow management systems towards workflow systems that are distributed and controlled in a truly decentralised manner. We build on this earlier work and consider how these patterns can be implemented in a decentralised environment with the help of the security protocols, and what additional patterns could be suggested for this type of system and operating environment. We discuss examples of these in section 3.

1.3 Roadmap

The rest of this article has the following structure. In section 2 we describe contemporary distributed security protocols that are essential in extending the application of workflow systems to a distributed processing model, and outline the message exchanges required by these protocols using sequence diagrams. In section 3 we discuss the implementation of distributed, decentralised workflows by analysing a selected set of recognised workflow patterns.

2 Security Protocols

Over the years many promising distributed security protocols have been proposed for handling a multitude of situations requiring participant collaboration in achieving a shared goal. We introduce the following protocols for building distributed workflow systems:

A group signature scheme Enables a group to sign messages in a way that a subset or all members of the group are required to provide their signature for the group signature to be valid. Later no member can deny their participation in providing this group signature.

A group safe scheme Distributes a secret between the members in a way that no member can reproduce the data without the others. If one member loses their share of the encryption key or their personal master key, this does not compromise the shared secret.

A group coin toss protocol Enables the participants to make a choice, and commit to these without being able to change their selection between the commitment and revealing the choice. This has a number of uses, and can be used for example in implementing an anonymous group communication protocol.

A distributed Byzantine agreement protocol Provides a mechanism for agreeing on proposed values between the participants. This can be used to implement distributed data replication schemes and distributed notice board services.

A Conference key protocol Enables the confidentiality of message exchange in a group. Lastly, anonymous broadcast protocols enable a group member to broadcast a message without revealing the sender's (or receivers') identity.

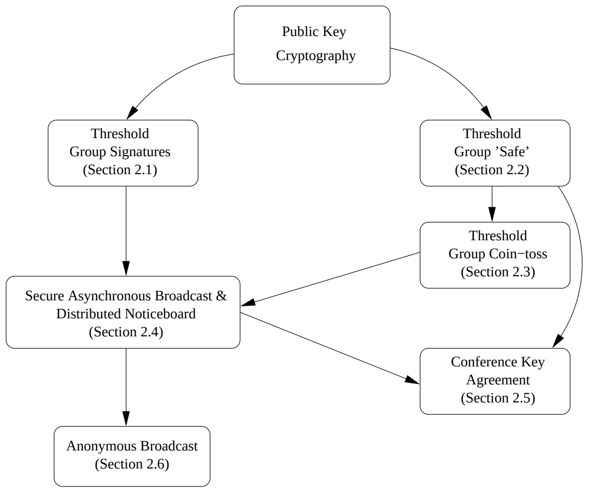

Public Key Cryptography
Threshold Group Signatures (Section 2.1)
Threshold Group 'Safe' (Section 2.2)
Threshold Group Coin−toss (Section 2.3)
Secure Asynchronous Broadcast & Distributed Noticeboard (Section 2.4)
Conference Key Agreement (Section 2.5)
Anonymous Broadcast (Section 2.6)

Figure 1: Overview of reviewed security protocols and their interdependencies

The relationships between these protocols are depicted in figure 1. For each of the protocols we will describe the problem they have been created to solve, provide an overview of the solution, show the message exchange between the participants, and give an example of a situation where the protocol can be used. We focus only on showing the message exchange of the protocols rather than discussing the cryptographic algorithms required to construct and process the message contents, as our focus is on discussing how these protocols can be implemented in a workflow environment (see section 3).

We are only able to discuss a small set of useful security protocols here. New developments in the security protocol field will continue to provide us with tools that will enable us to perform activities in a more efficient way, or enable us to do entirely new things. For instance, we have not discussed *Homomorphic encryption* protocols [7], where an activity can be performed on encrypted content and later reveal the result by decrypting the content, for example, we can add numbers to an encrypted value, without having knowledge of the value. However, if the encrypted sum is decrypted, the actual sum of the two is revealed. Homomorphic encryption has been used in creating secure voting systems, collision-resistant hash functions, and private information retrieval schemes.

2.1 Threshold group signature

In a group signing situation each of the participating signatories provides an individual signature; however, the overall signature is not valid unless the combined signature of all signatories is valid. A group signature scheme enables a group of participants to demonstrate the authenticity of messages by enabling a verifier to construct a composite signature from the *signature shares* provided by each signing party, and by enabling this composite signature to be verified by one public key pre-assigned to the group of signatories. In a *(k,l)-threshold signature* scheme at least *k* of the *l* parties are required for signing.

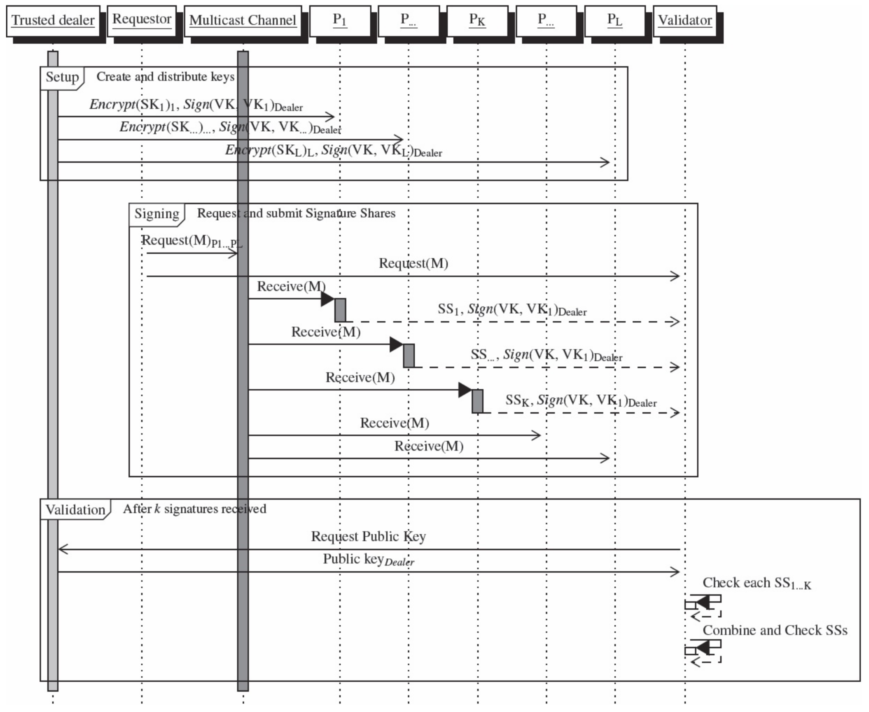

Trusted dealer
Requestor
Multicast Channel
P_1
$P_{...}$
P_K
$P_{...}$
P_L
Validator
Setup
Create and distribute keys
$Encrypt(SK_1)_1$, $Sign(VK, VK_1)_{Dealer}$
$Encrypt(SK_{...})_{...}$, $Sign(VK, VK_{...})_{Dealer}$
$Encrypt(SK_L)_L$, $Sign(VK, VK_L)_{Dealer}$
Signing
Request and submit Signature Shares
$Request(M)_{P1...PL}$
Request(M)
Receive(M)
SS_1, $Sign(VK, VK_1)_{Dealer}$
Receive(M)
$SS_{...}$, $Sign(VK, VK_1)_{Dealer}$
Receive(M)
SS_K, $Sign(VK, VK_1)_{Dealer}$
Receive(M)
Receive(M)
Validation
After k signatures received
Request Public Key
Public key_{Dealer}
Check each $SS_{1...K}$
Combine and Check SSs

Figure 2: Message exchange in the threshold signature protocol, where *SK = Key Share, VK = Verification Key, M = Message to Sign, and SS = Signature Share*

Figure 2 depicts the participating parties and the key message exchange for a threshold signature protocol described in [23]. In the first stage, the trusted dealer provides each participant with their key shares. In the signing stage, each participant that agrees to sign the document forwards their individual signature shares for validation of the group signature. A group signature scheme could be used to sign an agreement by multiple parties over an asynchronous messaging platform. Threshold signatures are required in section 2.4 to implement a security asynchronous broadcast and a distributed noticeboard protocol. These applications also require a distributed threshold coin-tossing protocol, which we will review in Section 2.3.

2.2 Threshold group *'safe'*

How can some secret data be shared among a number of participants in a way that no one will have control of the data over the other group members? We could store multiple copies of the secret, but this could result in a leak if one of the members is compromised. Similarly to the threshold signatures, some secret data D can be divided into n pieces in such a way that for any number k or more D_i pieces will make D easily computable, but any less than k pieces will leave D incomputable [21]. In other words, k sets the threshold of how many parties are required to co-operate to recreate the data. This threshold can be set when the protocol is initiated for each data item.

This protocol enables the implementation of a distributed *safe*, where no single party owns the entire set of data, and where k members are required to compose the data together. In other words, a group of mutually suspicious participants must cooperate. This is useful, for example, in storing and sharing secret keys among group members.

Figure 3 show the protocol messages and participants. An interesting example of the use of the protocol is the implementation of hierarchical organisation structures by, for example, giving the company's president three values of D_i, each vice-president two values of D_i, and each executive one value of D_i. In a threshold data safety scheme that requires threeout of N data shares, this arrangement enables documents to be signed by the president alone, by two of the vice-presidents or by three of the executives. This secret-sharing protocol provides a convenient building block for a distributed coin-tossing protocol, which we will discuss in the next section.

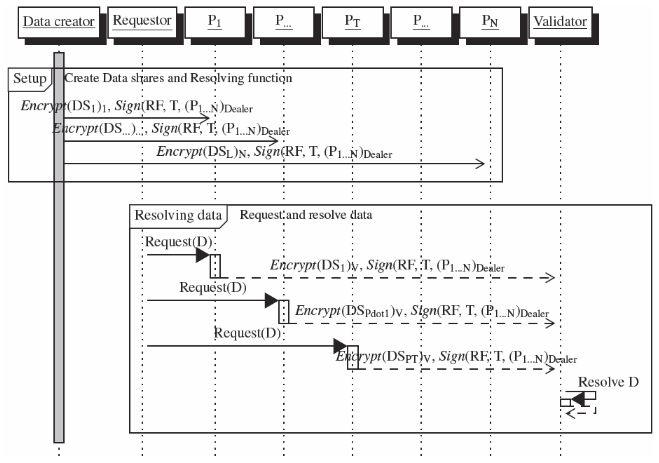

Data creator
Requestor
P1
P...
PT
P...
PN
Validator
Setup
Create Data shares and Resolving function
Encrypt(DS1)1, Sign(RF, T, (P1...N)Dealer
Encrypt(DS...)..., Sign(RF, T, (P1...N)Dealer
Encrypt(DSL)N, Sign(RF, T, (P1...N)Dealer
Resolving data
Request and resolve data
Request(D)
Encrypt(DS1)V, Sign(RF, T, (P1...N)Dealer
Request(D)
Encrypt(DSPdot1)V, Sign(RF, T, (P1...N)Dealer
Request(D)
Encrypt(DSPT)V, Sign(RF, T, (P1...N)Dealer
Resolve D

Figure 3: Message exchange in the threshold group safe protocol, where *DS = Data Share, RF = Resolving Function, and T = Threshold*

2.3 Threshold group coin toss

The problem is how to toss a coin between two parties who are not physically together, and prevent either party from lying about the result. The problem can be solved if the coin toss result can be stored in a way that can not be altered by the party that tossed the coin. The threshold group safe scheme discussed in the previous section is a suitable tool for this purpose, and Cachin et al. [4] build their solution on this protocol. The key of the protocol is based on a hash function scheme, where the value of a coin C $\{0,1\}$ is hashed with H to a value $\tilde{g}_0$ in group G of large prime order q. This value is then raised to a secret exponent x_0 to obtain $\tilde{g}_0$. Hash *H'* exists for testing the validity of the secret share validity, and finally a hash *H"* exists for restoring the coin value by calculating $H''(\tilde{g}_0)$.

The protocol message exchange is shown in figure 4 in generalised format. An actual coin toss can be achieved if both parties are required to access the 'safe' to retrieve the result. We use this protocol in section 2.4 to implement a distributed anonymous notice board. Other examples of use includes online games, where the parties must commit to a selection independently prior any party revealing their choice to others. This is useful when playing cards. Equipped with a group threshold signature, secret sharing and coin-tossing schemes, we can now proceed to the secure distributed broadcasting protocol.

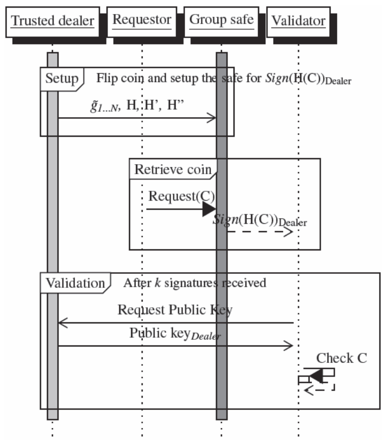


Figure 4: Message exchange in the threshold coin-toss protocol, where $\tilde{g}_{1...N}$ = *individual secret shares, H, H', H'' = Hash functions, and C = Coin*

2.4 Secure distributed agreement

Broadcast protocols are a fundamental building block for providing replication in distributed systems. Secure asynchronous protocols are needed, for example, in distributing *conference keys* among the group members, and publishing information about available services and public keys. A system where the broadcast information is stored prior to consumption, provides a distributed notice board service.

The concept is analogous to that of a distributed *Byzantine Agreement* [14], as the participating members must be able to agree on the correct value of particular information, store this and have a way of retrieving it afterwards without being subject to maliciously misbehaving participants trying to distribute any misinformation. We approach a solution via the work presented in [4]. Figure 5 shows the message exchange from the perspective of one participant P_i. In the pre-processing round all participants distribute their signature share to other participants. In the following steps the parties establish what value is being voted for, and then in subsequent voting rounds look for a winning result. All steps need to meet the pre-defined thresholds in order to avoid malicious parties to collaborate. A system where the broadcast information is stored prior to consumption provides a distributed notice board service. This can be implemented with the Asynchronous Byzantine Agreement protocol by using the protocol to vote on the correctness of the information offered to a requestor by one of the data holders.

2.5 Conference key agreement

In a conference key agreement protocol the participants together establish a common conference key to enable secure exchange of messages between the participants. Many conference key protocols have been proposed in the past with various degrees of proof for security [19, 3, 12, 15]. The most common approach in these protocols is again based on the idea of group secret sharing (see section 2.2), where

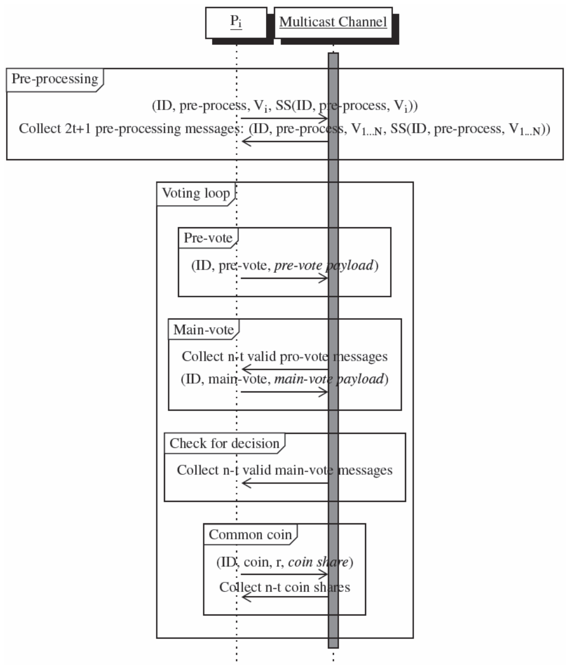

Pi
Multicast Channel
Pre-processing
(ID, pre-process, Vi, SS(ID, pre-process, Vi))
Collect 2t+1 pre-processing messages: (ID, pre-process, V1...N, SS(ID, pre-process, V1...N))
Voting loop
Pre-vote
(ID, pre-vote, pre-vote payload)
Main-vote
Collect n-t valid pro-vote messages
(ID, main-vote, main-vote payload)
Check for decision
Collect n-t valid main-vote messages
Common coin
(ID, coin, r, coin share)
Collect n-t coin shares

Figure 5: Asynchronous Byzantine Agreement for party P_i, where V_i = Initial Value, SS = Signature Share

a trusted dealer initialises the participants with individual key shares and resolving functions. Equipped with these, the participants can each generate a portion of the conference key and share their shares between each other. Concatenation of these shares becomes the conference key.

This is, of course, subject to any malicious activity from the participants, as they could send an invalid key share to one or more participants, and in this way exclude them from the conference. Huang et al. [11] have proposed a protocol that suggests a way to filter malicious participants at the beginning of the protocol to ensure that all participants obtain the same conference key.

Conference key agreement protocols are an active research area. Recently Harn and Lin [10] have proposed a protocol where a trusted dealer uses a secure broadcast, where the dealer or key generation center (KGC) broadcasts group key information directly to all participants. Another recent proposal was made by Zhao et al. [26], where they suggest a two-stage key creation process, where the initial conference key functions as a *master key* that is not used for data communications, but is used for creating *'session'* conference keys to be used for the actual data exchange. Their proposal provides *forward security*, which ensures that all previously issued session keys retain their secrecy even if any participant's master key would be compromised.

The conference key protocols provide the basic data confidentiality for data exchange between group participants analogously to how SSL session keys are created for one-to-one client-server connections based on the previously distributed identity certificates.

2.6 Anonymous broadcast

The protocol is based on P^5 (Peer-to-Peer Personal Privacy) protocol [22], which can be used to provide sender-,receiver-,and sender-receiver anonymity. The protocol is based on the creation of hierarchical of broadcast channels, where different levels of hierarchy provide different levels of anonymity. The channels are based on an overlay network structure, where the nodes are named based on a hash function of their real identity, and assigned to different segments of the hierarchy based on their pseudonym. Figure 6 shows how the broadcast channel relays messages. The key here is that the relaying node does not know if the sender is the message originator or just another relay-step. To prevent the traceback of messages by observing the traffic in the entire network, the participants are required to send noise messages and to mix the order of received messages before relaying them.

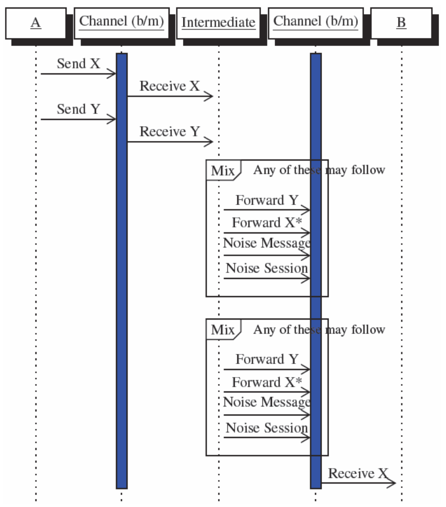


Figure 6: P^5: message X forward sequence to the destination channel (* If $X_{\text{Sender}} \neq X_{\text{Receiver}}$ and $m_B \leq m_A$)

2.7 Distributed mutual exclusion protocol

The issue of allowing only a certain number of participants from accessing their critical sections simultaneously is a distributed mutual exclusion (mutex) problem. A number of distributed mutex protocols have been proposed in the literature. Recently Chaudhuri and Edward [5] have proposed a protocol based on tokens and a hierarchical network model, where the participants can enter their critical section if they have first gained the possession of the token. Their approach relies on the construction of local and global mutex request queues.

In our organisational email environment, these queues could be managed by the allocated leaders of the group, assigned to a dedicated queue manager, or distributed using the group threshold signature and safe protocols. We leave the specifics of this discussion to a later work.

3 Distribution and de-centralisation of workflow patterns

The workflow patterns as reviewed in [1] and [20] consider workflows in a centrally controlled workflow management environment. They outline 126 different patterns, and while it would be interesting to discuss all of these from the decentralisation perspective, we have selected a set of key patterns in order to discuss fundamental aspects of distributed workflow management, the application of the security protocols discussed in Section 2, and a set of more complex patterns that highlight some of the more challenging aspects of process distribution. We discuss the control-flow, data, and resource management perspectives of the patterns, and focus on the aspects that relate to management of task concurrency and co-ordination between participants – aspects that are normally managed by a central workflow management system, but require participant co-operation in a de-centralised setting. In the following sections we use the pattern numbering introduced in [1] and [20].

3.1 Control-flow patterns

The control-flow patterns describe tasks and how their execution is controlled via connecting constructors like sequence, choice, parallel execution and join. From the perspective of de-centralisation, interesting control-flow issues relate to aspects of concurrency management, and co-ordination between the participants. We first discuss the *Sequence* pattern, which provides the foundation for linking tasks into processes. The *Multiple instances without a priori runtime knowledge* show how different participants can co-operate in ensuring workflow structure, and the *Interleaved Parallel Routing* provides an example of a task sequence ordering between parties that are executing concurrent threads of a flow.

3.1.1 Pattern WCP-1 (Sequence)

The *Sequence* pattern provides the most fundamental building block for processes by enabling the sequential execution of tasks one after the other. Traditionally a CWMS tracks the progress of processes and manages the allocation of the *task instances* (work items) to resources (see Section 3.4), provides status information, and handles error situations. In a decentralised environment the participants need to co-operate in achieving the same functionality, for example, by using a group broadcast protocol for sending updates on process status.

All work item life cycle stages *(offered, allocated, withdrawn, started, completed, failed)* and transitions between these stages need consideration in the decentralised model. Who will maintain oversight of the work item orchestration? In our proposed SOE the instance that initiated the process for the service can perform this function, and as services are built on other services, the orchestration is a hierarchical responsibility. If a truly distributed orchestration service is required, the participating members can share the work item information via secure broadcast protocols and use existing scheduling algorithms to share the management load.

3.1.2 Pattern WCP-15 Multiple instances without a priori runtime knowledge

"Within a given process instance, multiple instances of a task can be created. The required number of instances may depend on a number of runtime factors, including state data, resource availability and inter-process communications and is not known until the final instance has completed. Once initiated, these instances are independent of each other and run concurrently. At any time, whilst instances are running, it is possible for additional instances to be initiated. It is necessary to synchronize the instances at completion before any subsequent tasks can be triggered." [20]

Task C in figure 7 is the multiple instance task. The two dashed areas (pre and post) show the areas of the CPN petrinet model that implement the concurrent instance creation and accounting, and the termination and merging chores respectively. While in a centrally managed workflow these chores would be performed centrally; in a decentralised setup the activity must be distributed. This chore can be split in a number of ways. For example, the pre-processing activity could be performed by a node that was allocated it by a workload balancing algorithm, and the post-processing activity could be performed

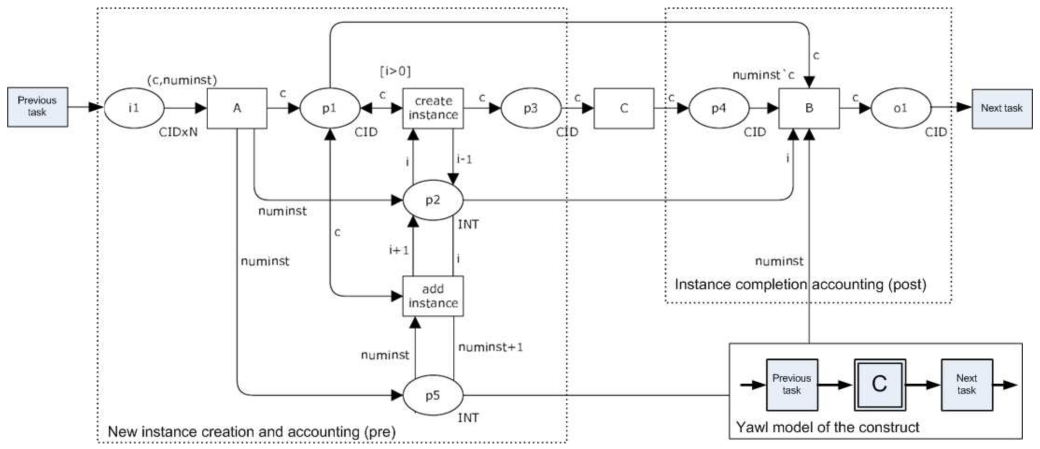


Figure 7: Pattern *WCP-15 Multiple instances without a priori runtime knowledge* together with its YAWL-model counterpart, adapted from a CPN-model in [20]

by the node that is allocated with the next task. Whether this is an acceptable arrangement would depend on whether the next task can be trusted with the task of merging the parallel threads, as they would be in a position to omit individual results from selected threads. Should this be an acceptable arrangement, the pre-processor and the post-processor are required to establish a secure communication channel, for example, by using a conference key agreement protocol, so that the pre-processor can safely pass the information from *p1*, *p2* and *p5* to the post-processor.

3.1.3 Pattern WCP-17 Interleaved Parallel Routing

"The Interleaved Parallel Routing pattern offers the possibility of relaxing the strict ordering that a process usually imposes over a set of tasks. Note that Interleaved Parallel Routing is related to mutual exclusion, i.e. a semaphore makes sure that tasks are not executed at the same time without enforcing a particular order." [20] In figure 8 the place p3 implements a mutex variable in the CPN-model of the pattern. In a de-centralised environment an equivalent implementation can be achieved using the distributed mutual exclusion protocol outlined in section 2.7.

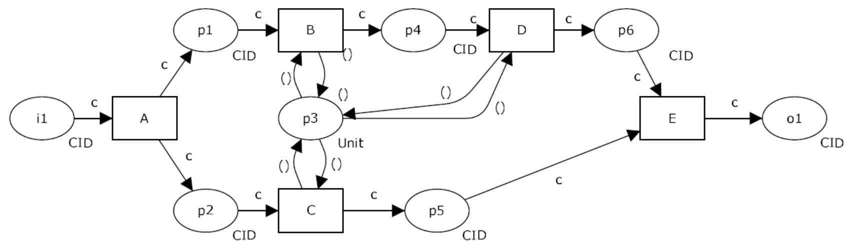


Figure 8: Pattern WCP-17 Interleaved Parallel Routing as depicted in [20]

3.2 Data patterns

Data are the second key ingredient in workflow processing; the control flows create, manipulate, transfer and store data as well as change their behaviour, for example, by making routing decisions based on the data values. Each one of these aspects provides interesting challenges in a decentralised workflow environment. We follow the visibility, interaction and transfer patterns defined in [20].

3.2.1 Data visibility patterns

The data elements can be defined and utilised at the different levels of the workflow process definition. The data can be confined to single tasks (WPD-1), accessed by all components of a sub-process (WPD-2) or by a defined set of tasks (WPD-3), shared by parallel multiple task instances (WPD-4 a) or block tasks (C & D) that share the same sub-process implementation (WPD-4 b), or accessed by all tasks within a workflow instance (WPD-5) (Figure 9). The data visibility scopes can be enforced by dynamically creating conference keys for the participants involved with the tasks in scope. Similarly to the data visibility scopes, group signature and anonymous broadcast channels can be dynamically created for the participants where these services are needed.

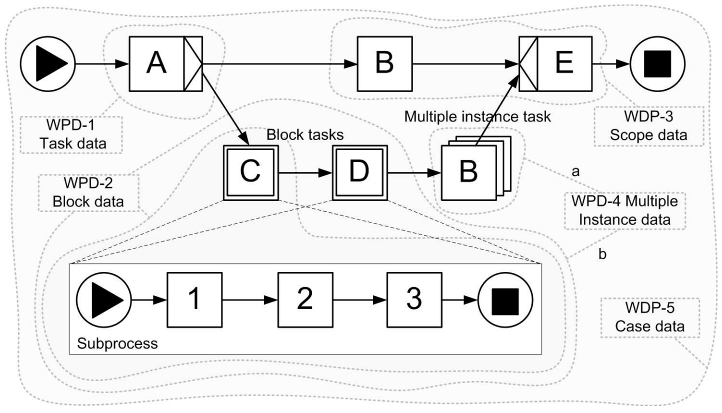


Figure 9: Summary of process data visibility patterns

3.3 Data interaction patterns and transfer patterns

Each of the data visibility patterns need to be supported by a mechanism to communicate the data elements between process components. Generally speaking, the data interaction can take place from task to task, from block task to sub-process and back, from task to multiple instance task and back, from process instances to instance, and from the external environment to process components and back. The data transfer mechanism in these different forms of interaction; can happen a number of ways, namely by moving or copying the data by value, or by reference with or without locking source location of the element.

In the case of the *SOE*, the data elements are transferred by email messages. Most of the messages can be handled automatically, with only the messages that require user actions displayed. However, the transfer of data elements does not need to follow the control flow of the workflow. For example, in the case of the WPD-2 pattern, *Task C* could pass all of the elements that *Tasks 1, 2 and 3* require to *Task 1*, which could then forward elements to *Tasks 2 and 3* via *Task 2*. Should some data not be required by and be kept confidential from *Tasks 1* and *2*, this could be encrypted with a shared key between *Tasks C* and *3*, but *Tasks 1* or *2* could still maliciously interfere with the data. Alternatively, *Task C* could send these data directly to *Task 3*.

Obviously, the most suitable approach for moving the data depends on the nature of the data and the trust levels and security clustering of the participating tasks. This demonstrates how the data-flow design must be designed in parallel with the controlflow design, and supported by integrated data-flow and security modeling tools. Figure 10 shows data flow behavior with control flow.

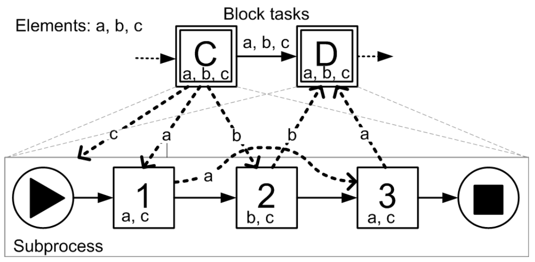


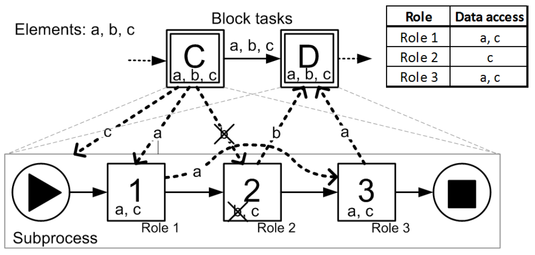
Figure 10: Data-flow semantics as part of a workflow model

3.4 Resource patterns

Resources that perform the tasks in the workflow processes are the third key ingredient to complete workflow definitions. Patterns from WRP-1 to WRP-39 [20] are also all feasible usage structures in decentralised processing environments. We discuss below three patterns that are relevant to our earlier discussion on security protocols.

3.4.1 Pattern WRP-2 Role-based distribution

The ability to specify at design time one or more roles to which instances of this task will be distributed at runtime. Roles serve as a means of grouping resources with similar characteristics. [20] The role-based task allocation goes hand in hand with the data visibility and transfer patterns when enforced with data encryption. For example, the role holders will also need to be assigned with the corresponding conference or secure broadcast channel keys.



Role	Data access
Role 1	a, c
Role 2	c
Role 3	a, c

Figure 11: Role data with Control and Data flow semantics as part of a workflow model

Combined with the control-flow and data-flow modelling, task role allocation modeling enables us to analyse and show inconsistencies between these different aspects of model definition (figure 11). In a decentralised workflow environment the interplay between the required data elements, allocated resources, and available new security services is complex, and requires integrated modelling tools for process design and monitoring.

3.4.2 Pattern WRP-13 Distribution by Offer - Multiple Resource

While the original WRP-13 pattern refers to a situation where a work item is offered to a number of suitable resources, of which one will accept the offer, the pattern could be extended to allow a group of resources to accept an offer to perform a block task. Considering the threshold group signature scheme (discussed in 2.1), the role-to-resource mapping should not only be considered as a way of grouping resources with similar characteristics. To return to our earlier example where the ability to sign a document depended on how many of the required signature shares can be pooled together by a group of users, this could introduce a different type of *team-role* pattern, where a block task can be allocated to a team of resources who together can perform the activities within the sub-process.

3.5 About our reference model implementation

The YAWL project [2] has produced a detailed CPN-tools based [18] simulation model for the *newYAWL*-workflow modeling language. Our research builds on this model. This includes a number of challenges. Firstly, the main workflow management framework must support the distribution of the *work distribution* function and the implementation of an asynchronous data distribution mechanism for shared variables required for the overall management of the system. We aim to implement minimal core functionality in the CPN-model, and the security, data distribution and mutual exclusion protocols as workflow definitions.

4 Conclusion

Distributed security protocols for group communications provide a mechanism for implementing data confidentiality, authenticity, non-repudiation and privacy for collaborative workflow environments. They enable a flexible approach where these security objectives can be met at different levels of granularity. We have reviewed a number of contemporary distributed security protocols, and discussed how these can be utilised to implement the security functionality expected of centralised workflow systems, as well as new functionality enabled by the very nature of de-centralisation. For example, from the perspective of privacy the distributed environment offers further benefits in the form of participant anonymity within a group as well as being able to act on behalf of a group rather than an individual participant.

References

[1] Wil M. P. van der Aalst, A. Hofstede & M. Weske (2003): *Business process management: A survey*. In: *2nd International Business Process Management Conference (BPM'2003)*, pp. 369–380.

[2] Wil M. P. van der Aalst & A. H. M. ter Hofstede (2005): *YAWL: yet another workflow language*. *Inf. Syst.* 30(4), pp. 245–275.

[3] Daniel Augot, Raghav Bhaskar, Valerie Issarny & Daniele Sacchetti (2005): *An Efficient Group Key Agreement Protocol for Ad Hoc Networks*. In: *Proceedings of the First International IEEE WoWMoM Workshop on Trust, Security and Privacy for Ubiquitous Computing*, IEEE Computer Society, pp. 576–580.

[4] Christian Cachin, Klaus Kursawe & Victor Shoup (2005): *Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography*. Journal of Cryptology 18(3), pp. 219–246.

[5] Pranay Chaudhuri & Thomas Edward (2008): *An algorithm for k-mutual exclusion in decentralized systems*. Computer Communications 31, pp. 3223–3235.

[6] Georgios John Fakas & Bill Karakostas (2004): *A peer to peer (P2P) architecture for dynamic workflow management*. Information and Software Technology 46, pp. 423–431.

[7] Caroline Fontaine & Fabien Galand (2007): *A survey of homomorphic encryption for nonspecialists*. EURASIP J. Inf. Secur. 2007, pp. 1–15.

[8] Li Guo (2006): *Enacting a Decentralised Workflow Management System on a Multi-agent Platform*. Ph.D. thesis, School of Informatics, University of Edinburgh.

[9] Li Guo, Dave Robertson & Yun-Heh Chen-Burger (2008): *Using multi-agent platform for pure decentralised business workflows*. Web Intelli. and Agent Sys. 6(3), pp. 295–311.

[10] L Harn & C Lin (2010): *Authenticated Group Key Transfer Protocol Based on Secret Sharing*. IEEE Transactions on Computers 99.

[11] Kuo-Hsuan Huanga, Yu-Fang Chungb, Hsiu-Hui Leed, Feipei Laia & Tzer-Shyong Chen (2008): *A conference key agreement protocol with fault-tolerant capabilit*. Computer Standards & Interfaces 31, pp. 401–405.

[12] Bae Eun Jung (2006): *An efficient group key agreement protocol*. IEEE communications letters 10.

[13] Petteri Kaskenpalo (2009): *Service oriented email for organisation modeling and process execution*. In: Proceedings of the 1^{st} International Workshop on Organizational Modeling (ORGMOD 2009).

[14] Leslie Lamport, Robert Shostak & Marshall Pease (1982): *The Byzantine Generals Problem*. ACM Trans. Program. Lang. Syst. 4(3), pp. 382–401.

[15] Patrick P. C. Lee, John C. S. Lui & David K. Y. Yau (2006): *Distributed collaborative key agreement and authentication protocols for dynamic peer groups*. IEEE/ACM Trans. Netw. 14(2), pp. 263–276.

[16] Edward J. Lusk (2006): *Email: Its decision support systems inroads–An update*. Decision Support Systems 42(1), pp. 328–332.

[17] Luke McDowell, Oren Etzioni, Alon Halevy & Henry Levy (2004): *Semantic email*. In: WWW '04: Proceedings of the 13th international conference on World Wide Web, ACM Press, New York, USA, pp. 244–254.

[18] Anne Vinter Ratzer, Lisa Wells, Henry Michael Lassen, Mads Laursen, Jacob Frank, Martin Stig Stissing, Michael Westergaard, Sren Christensen & Kurt Jensen (2003): *CPN Tools for editing, simulating, and analysing coloured Petri nets*. In: Applications and Theory of Petri Nets 2003: 24th International Conference, ICATPN 2003, Springer Verlag, pp. 450–462.

[19] RSA Laboratories (1993): *PKCS# 3: Diffie-Hellman Key-Agreement Standard*. Technical Report.

[20] Nicholas Charles Russell (2007): *Foundations of process-aware information systems*. Ph.D. thesis, Faculty of Information Technology, Queensland University of Technology.

[21] Adi Shamir (1979): *How to share a secret*. Commun. ACM 22(11), pp. 612–613.

[22] Rob Sherwood, Bobby Bhattacharjee & Aravind Srinivasan (2005): *P^5: A protocol for scalable anonymous communication*. J. Comput. Secur. 13(6), pp. 839–876.

[23] Victor Shoup (2000): *Practical Threshold Signatures*. In: Advances in Cryptology EUROCRYPT 2000.

[24] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski & A. P. Barros (2003): *Workflow Patterns*. Distrib. Parallel Databases 14(1), pp. 5–51.

[25] Jun Yan (2004): *A framework and coordination technologies for peer-to-peer based decentralised workflow systems*. Ph.D. thesis, Swinburne University of Technology.

[26] Jianjie Zhao, Dawu Gu & Yali Li (2010): *An efficient fault-tolerant group key agreement protocol*. Computer Communications 33, pp. 890–895.

Separating Agent-Functioning and Inter-Agent Coordination by Activated Modules: The DECOMAS Architecture

Jan Sudeikat[*]and Wolfgang Renz
Multimedia Systems Laboratory (MMLab),
Faculty of Engineering and Computer Science, Hamburg University of Applied Sciences,
Berliner Tor 7, 20099 Hamburg, Germany
{jan.sudeikat|wolfgang.renz}@.haw-hamburg.de

The embedding of *self-organizing* inter-agent processes in distributed software applications enables the decentralized coordination system elements, solely based on concerted, localized interactions. The separation and encapsulation of the activities that are conceptually related to the coordination, is a crucial concern for systematic development practices in order to prepare the reuse and systematic integration of coordination processes in software systems. Here, we discuss a programming model that is based on the externalization of processes prescriptions and their embedding in *Multi-Agent Systems* (MAS). One fundamental design concern for a corresponding execution middleware is the minimal-invasive augmentation of the activities that affect coordination. This design challenge is approached by the *activation* of agent modules. Modules are converted to software elements that reason about and modify their host agent. We discuss and formalize this extension within the context of a generic coordination architecture and exemplify the proposed programming model with the decentralized management of (web) service infrastructures.

1 Introduction

Self-Organization describes adaptive processes among system elements, as found in physical, biological, and social systems, that establish and maintain structures [19]. The utilization of self-organization principles is an alternative approach for the construction of self-adaptive, distributed software systems [23]. This approach is attractive, as it allows to embed adaptive properties in the interplay of system entities. Consequently, centralized responsibilities are avoided that may imply bottle necks and single points of failure. Self-organizing system phenomena are governed by feedback loops, i.e. circular interdependencies among system elements (e.g. discussed in [1]). Unlike the control loops in self-managing software systems [5], the loops are decentralized, i.e. distributed among system elements.

In the research project ”Selbstorganisation durch Dezentrale Koordination in Verteilten Systemen”[1] (Sodeko VS), the utilization of self-organizing inter-agent processes as reusable design elements is studied [29]. Distributed feedbacks, as structures of mutual influences among system elements, are elevated to discrete design elements. These structures are used to define inter-agent processes and a corresponding programming model can be used to integrate these processes in agent-based software systems. A foundational building block is a middleware layer that provides an execution context for the process enactment and integration [29] (see Section 3).

A key design criterion is that adaptive features can be supplemented to functioning sets of software agents, i.e. *Multiagent Systems* (MAS). Developers can add decentralized coordination, i.e. the

[*]Jan Sudeikat is doctoral candidate at the Distributed Systems and Information Systems (VSIS) group, Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, University of Hamburg, Vogt–Kölln–Str. 30, 22527 Hamburg, Germany, jan.sudeikat@informatik.uni-hamburg.de

[1]Self-Organisation by Decentralized Coordination in Distributed Systems

self-organization of system aspects, to working systems. A prerequisite is the conceptual and practical separation of the activities that are conceptually related to the inter-agent coordination. These activities concern the participation in a coordinating process and define a supplement, which influences the core functionality of the agents. In this paper, we present an approach for this separation that is based on extending agent-oriented implementation modules. This allows the minimal-intrusive encapsulation and automation of inter-agent coordination. In addition, the discussed enhancement is attractive for MAS developers as it allows to modularize crosscutting concerns in MAS (see Section 2.2).

This paper is structured as follows. In the following section, related work is outlined. In Section 3, a programming model for self-organization is outlined. The technological foundation for the encapsulation and automation of coordination activities is the activation of agent modules that is discussed in Section 4. The utilization of the programming model is exemplified in Section 5 before we conclude and give prospects for future work (see Section 6).

2 Related Work

Agent technology provides tools and concepts for the construction of autonomous software elements and is a prominent grounding for the development of self-organizing applications [25]. Natural self-organizing systems are composed of *autonomous* system elements [19], e.g. particles and cells, and the coaction of these elements can be metaphorically resembled with autonomous software agents.

2.1 Integrating Coordination Mechanisms

The construction of self-organization, i.e. an adaptive, coordinating process that structures the configurations of system elements, is based on two foundational types of implementation mechanisms [31]. First, generic *interaction*-level mechanisms have been proposed that allow to establish information flows between system elements (e.g. reviewed in [8]). Among others, these mechanisms support the stochastic dissemination of information and the attenuation of outdated data. Secondly, adaptation-level mechanisms control the participation in interactions, the processing of the exchanged information, and affect the conclusive adjustments within software agents that result from the exchanged information.

The encapsulation of interaction-level mechanisms is typically approached by dedicated communication infrastructures and languages [12]. These are means to decouple software components but the coordination logic, e.g. when to interact and how to (locally) respond to interactions is blended in the control flow the system elements. In previous works, three foundational approaches have been followed to separate this control of the coordination from the control of the element functioning. First, specialized agent architectures (e.g. [27]) have been proposed that outsource coordination-related activities to specific *modules*. Secondly, the separation can be enforced by the *execution infrastructure* that is given by the utilized programming-language or middleware. An example is the outsourcing of coordination by using aspect-orientation [24]. Finally, approaches use networked elements to control the localized adjustments [26].

In this paper, a coordination middleware layer is presented that extends the modularization-based approaches (see above) to enable the separation and integration of coordination logic in *generic* agent architectures. Preparing the integration in established, general-purpose agent-architectures allows to reuse the existing constructive knowledge and tool support that concerns these agent models, e.g. methodology-specific agent design techniques. The direct integration, by reusing agent-modularization concepts avoids the communication overhead of externalized approaches.

2.2 Crosscutting Concerns in Agent-Orientation

Modularization enforces the decomposition of software systems into functional clusters with minimum overlapping functionality. These so-called *core concerns* are typically separated into different components or modules [17], complex software systems often comprise additional *crosscutting concerns*, so-called *aspects* [15], which are to be referenced from various modules. Prime examples are amongst others *failure recovery*, *monitoring* and *logging*. While these functionalities can be clustered in modules, these will be referenced throughout the agent model. Thus the information when to invoke the functionality is spread and references are scattered. In this respect, the embedding of coordination is regarded as another crosscutting concern. When the logic how to adjust and interact, as to participate in a collaborative process, is encapsulated in specific module, the contained activities have to be frequently referenced in the agent model and these references with be scattered as well. Consequently, it is desirable to contains the functionality, as well as the context if it's invocation in a single agent element.

The notion of *crosscutting* concerns for agent modularization has to date found minor attention. Numerous MAS infrastructures are build with object-oriented programming languages, thus Aspect-oriented Programming (AOP) [15] is one approach to embed crosscutting functionalities with additional programming language tool sets. In [16], it is exemplified how AOP techniques can be used to modularize object-oriented agent models by encapsulating mobility related API calls that are available in the JADE[2] agent platform and Garcia et al. [9], examined how AOP frameworks facilitate the realization of object-oriented agents. Examples of aspects in software agents are *interaction*, *mobility* and *learning* [10].

2.3 Agent Modularization Using the Example of BDI Agents

Agent platforms [2] provide distributed middlewares for the construction of MAS. One prominent architectural model is the *Belief Desire Intention* (BDI) architecture [20] that allows to express both longterm goal-directed objectives as well as reactivity. Following this architectural style, agents are structured as sets of *Beliefs*, *Goals*, and *Plans*. Beliefs contains the local knowledge of the agent about itself and the environment. Goals represent the objectives and plans are the executable means of agents. BDI-specific reasoning engines control the agent execution. The currently active goals are deliberated and means-end reasoning is used to select plans for the achievement of goals. Modularity in terms of functional independent clusters has been introduced to BDI agents by the *Capability* concept [6, 4]. Capabilities describe clusters of BDI concepts, i. e. Beliefs, Goals and Plans in a name-space. These enable the recursive inclusion of other capabilities (sub-capability). The interplay with a surrounding agent/capability (super-capability) is controlled by scoping rules. The visibility of the comprised elements is specified as well as the visibility of relevant events, generated outside of the capability.

In [22], a goal-centric modularization scheme, so-called *Goal-Oriented Modularity*, has been proposed. Modules encapsulate the information how sets of related goals can be satisfied. This enables a higher degree of encapsulation of behaviors.A behavior-based stance towards agent modularization is given in [7] where *role* concepts encapsulate sets of beliefs, goals, plans and reasoning rules. In addition, the *Enactment* and *deactment* of roles at run-time is prepared. Modularization by *policy-based intentions* is proposed in [13]. Developers can explicitly declare in which context a module is to be activated.

Due to the widespread, recognition, and practicability of the BDI agent model, the prototype realization of the here discussed coordination middleware (see Section 3) is based on this agent type. In this paper, the utilized enhancements to modularize *crosscutting concerns* are discussed in general (see Section 4) and are then detailed for this particular agent model (see Section 4.1).

[2]http://jade.tilab.com/

3 The DECOMAS Architecture

Within the SodekoVS research project [29], a programming model for distributed feedbacks among system elements is revised. A key objective of this framework is that the ability to self-organize is provided as an optional tool that development teams can integrate in their applications when needed. The aim is to enable the supplementation of self-organizing properties when the need for decentralized coordination of system elements is revealed. The foundational elements are a declarative configuration language [33] and an architectural model for the supplementation of externally prescribed inter-agent processes. Here, the configuration of self-organizing processes is not discussed. Details on the configuration language can be found in [33] and a graphical representation of the process description is exemplified in the Sections 5.1 and 5.2 to illustrate the intended application dynamics.

This integration architecture follows a layered structure that is illustrated in Figure 1. The *Application Layer* contains the application functionality. Within this layer, agents act as providers of application-dependent functionalities. An underlying *Coordination Layer* controls the enactment of coordinating processes among (sub-)sets of agents. *Coordination Media* are conceptual entities that contain interaction mechanisms [8]. Inside these media, these are realized by the utilization of communication infrastructures (e.g. see [12]). The details of the interaction mechanisms are hidden by a generic publish/subscribe interface. The utilization of these media is shielded from the agent internals by intermediate *Coordination Endpoints*. These are associated to an agent and control the participation in a coordination process. Endpoints are enabled to observe and modify the execution of associated agents. The rationale is that Endpoints interact, via Media, in place of the associated agent and decide the local adjustments. Therefore, coordination-relevant activities, including adaptation-level mechanisms, are encapsulated (see Section 2.1). The operation of Endpoints are declaratively configured (see above). These declarations indicate which changes in the agent-internal configuration are significant for their participation in an inter-agent process. These events are then propagated via Media and processed by perceiving Endpoints. If these perceptions indicate the need for adjustments, these are made by triggering agent-internal behaviors. Agent models are often capable to show concurrent conduct of behaviors and their scheduling is realized in the agent execution environment (e.g. see Section 4.1.1). Agents can be associated to more then one Endpoint, so the enactment of different processes is separated as well. A prototypical implementation of this architecture is reported in [32] and it's utilization is exemplified in [34].

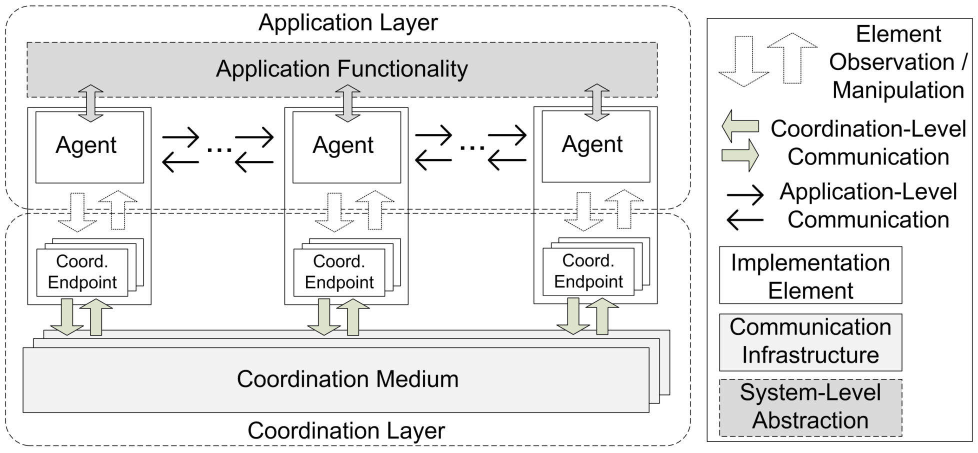


Figure 1: The SodekoVS-Architecture to the embedding of decentralized coordination in MAS [34].

4 Activating Agent Modules: Enabling Contributive Processing

Agent-oriented software development is supported by comprehensive development environments. These provide execution middleware and programming languages for the utilization of agent-based implementation concepts [2]. It is desirable that agent developers can utilize these concepts throughout the whole development cycle, also when expressing cross-cutting concerns (see Section 2.2). Conventional modules cluster functional concerns. These are typically used inside agents by explicitly referencing contained elements, e.g. dispatching (sub-)goals that are contained modules [4]. The aim of the proposed extension is to automate these references. Both the functionality and the information when it is to be invoked are contained in modules. These modules extend conventional agent modules, as modules are equipped with the ability of observe and modify the agent execution. These enhanced modules operate as autonomous actors that react to changes in the immediate context, i.e. the state of their host agent or super module. We name these modules *co-efficient*, since they register for contributive processing on certain agent reasoning events. The presented module concepts allows to compose agents as sets of independent actors. Besides the structuring of agent models, these modules facilitate the embedding of crosscutting mechanisms, like logging, failure recovery etc., to be automatically triggered. A example is the encapsulation of the monitoring of agent-behaviors in [30]. A module observes the reasoning of the host agent and decides the recording, when the course of action is significantly adjusted.

Figure 2 illustrates the conceptual model of a co-efficient agent module. Abstracting from specific agent architectures, we assume that the execution of an *Agent Model* is managed by a *Reasoner*, e.g. using reactive or deliberative mechanisms. The reasoning component processes *Agent Reasoning Event*s that characterize the agent execution and reference agent elements which are modified. Examples are changes in agent-intern data structures (knowledge) or the execution of plans. A module concept allows to structure agents by containing sets of agent elements (e.g. [6]). Co-efficient modules extend these with two additional components. First, an *Observation / Adjustment Component* allows to observe and modify agent execution by registering for and dispatching reasoning events. This component makes use of platform-specific interfaces (*Observation / Inducement*). Secondly, developers specify an *Event Mapping* that describes which events are subject of observation as well as which events are to be dispatched to the reasoning mechanism.

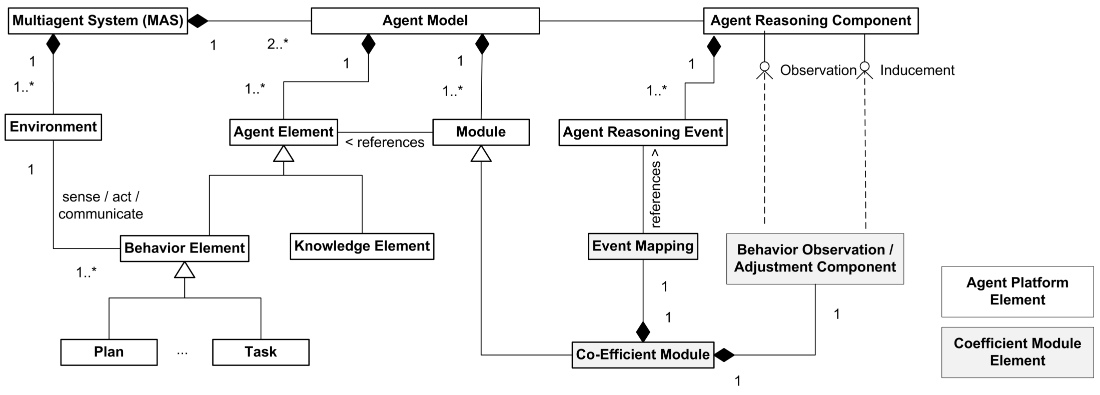


Figure 2: Conceptual model of co-efficient agent modules.

Figure 3 outlines the operating principle of a coefficient module. On agent start-up the module is registering (*Observation* interface) for the observation of a set of reasoning events in the surrounding agent (1). Subsequently, it is notified about these events (2) and the event mapping is interpreted to infer which events to dispatch via the *Inducement* interface. These events reference agent elements (e.g. goals, beliefs) inside the module (3), or in the surrounding agent (4). The available reasoning events can be classified according to their effects on the agent execution. Events denote modifications of the agent state, the agent behavior, the agent model, or describe communicative activities. The concrete realization of these event categories depend on the utilized agent platform and architecture (e.g. see section 4.2).

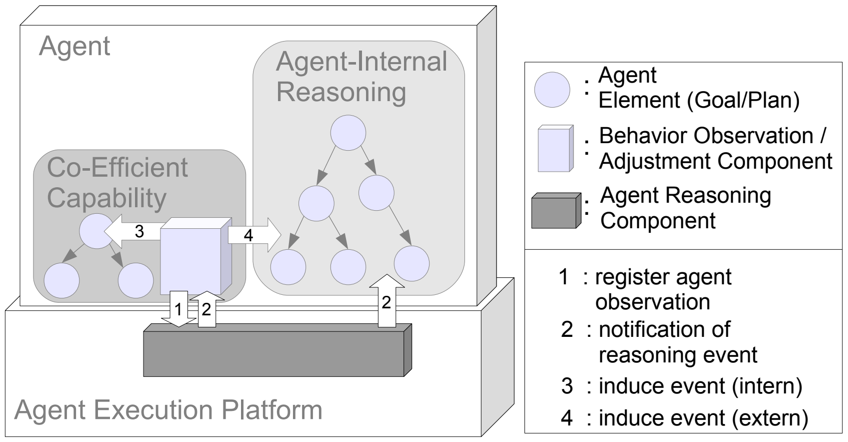

Agent
Agent-Internal Reasoning
Co-Efficient Capability
3
4
1
2
2
Agent Execution Platform
: Agent Element (Goal/Plan)
: Behavior Observation / Adjustment Component
: Agent Reasoning Component
1 : register agent observation
2 : notification of reasoning event
3 : induce event (intern)
4 : induce event (extern)

Figure 3: Execution of a Co–Efficient Capability.

4.1 Formalization of Coefficiency in Operational Semantics

In [20] the BDI architecture has been defined by an abstract interpreter and a corresponding proof theory. This seminal work bridges the gap between the theory and practice of BDI agents and led to the logic–based *Agentspeak(L)* programming language [21]. Based on this language and further formalizations, the Operational Semantics for BDI agents have been given in [35] to support implementation and verification of BDI-based MAS. Here, we adopt these semantics to formally express the impact of co-efficient modules. Within BDI agents these are Coefficient Capabilities (CECs) (see Section 2.3)).

4.1.1 Operational Semantics for BDI Agents

Operational Semantics are an established formalism to describe the semantics of programming languages. The operation of programs is expressed by a transition relation between program configurations [18]. The complete specification of the Operational Semantics of BDI reasoning is given in [35]. In this section, key definitions are summarized that are used to specify the effects of crosscutting concerns on the execution of BDI agents (cf. Section 4.1.2).

An agent configuration is defined as a a tuple $< ag, C, M, T, s >$ [35]. ag is the agent program, given by a set of *beliefs* and *plans*. C is a *Circumstance* that resembles the execution context of an individual agent. M is a tuple that characterizes the agent communication. T is a tuple that provides temporary information that is used by the reasoner and s is the current step in the reasoning cycle.

In the following, these elements are detailed. The Circumstance C is defined as a tuple $< I, E, A >$, where the element I is a set of *intentions* $\{i, i', \ldots\}$. An intention i is a stack of partially instantiated plans. E is a set of *events* $\{(te, i), (te', i'), \ldots\}$. These are denoted as pairs (te, i) of a triggering event (te) and a related intention (i). When events result from the processing of other events, e.g. from the achievement of (sub)goals, these follow-up events are associated to the currently active intention. An empty intention is denoted by $\top$. In this respect, intentions represent different courses of actions. Their concurrent execution is controlled by the agent reasoner [35]. A is a set of actions that is available to the agent to modify the environment.

The asynchronous communication of agents is characterized by the tuple M. The communicative abilities of agents are not influenced by CECs, therefore the management of communications is not discussed. Details can be found in [35].

Temporary information is kept in the tuple T. The elements $< R, Ap, \iota, \rho, \varepsilon >$ provide volatile data to the reasoner. The set R is the set of plans that are *relevant* for the current event. These plans are capable to handle the event. Ap denotes the set of plans that are not only relevant but can be activated. The elements ι, ε, ρ refer to the current intention, event and applicable plan that are considered during one reasoning cycle.

Finally, the current step in the reasoning cycle is given by the element s. Altogether the reasoning cycle is composed of nine steps $s \in \{ProcMsg, SelEv, RelPl, ApplPl, SelAppl, AddIm, SelInt, ExecInt, ClrInt\}$. These steps are: processing incoming messages, selecting an event to be handed, computing the relevant plans, computing the applicable plans, adding means, i.e. plans, to an intention, selecting an intention, executing an intention, clearing an intention [35].

CECs affect the selection of the handled events. Therefore, we summarize here the semantics of the original *Event Selection* rule (*SelEv*). This rule picks an event and marks it for further processing. BDI agents employ *reactive planning* they handle the events in E. Events are added to E by transition rules or elements of the general architecture outside the agent interpreter, e. g. belief updates. The rule SelEv1 refers to the *selection function* S_E that selects events from the set E. Selected events are removed from E and added to ε for further processing. If no event is to be handled, the rule SelEv2 skips directly to the intention execution that is initialized by the selection of an intention (SelInt) [35]:

$$\textbf{SelEv1} \quad \frac{S_E(C_E) = \langle te, i \rangle}{\langle ag, C, M, T, SelEv \rangle \longrightarrow \langle ag, C', M, T', RelPl \rangle} \tag{1}$$

$$\begin{aligned} where: C'_E &= C_E \setminus \{\langle te, i \rangle\} \\ T'_\varepsilon &= \langle te, i \rangle \end{aligned}$$

$$\textbf{SelEv2} \quad \frac{S_E(C_E) = \{\}}{\langle ag, C, M, T, SelEv \rangle \longrightarrow \langle ag, C, M, T, SelInt \rangle} \tag{2}$$

In order to handle selected events, the relevant and applicable plans are calculated and one applicable plan is selected. A group of rules is responsible to execute this applicable plan. Additional rules control the execution of different intentions. *External* events, i. e. events that are perceived and not generated by previous plan executions trigger the creation of a novel intention. These stacks of partially instantiated plans are added, removed, and selected for execution by dedicated transition rules [35].

4.1.2 The Operational Semantics for Co-Efficient Capabilities

Co–efficient modules register themselves for agent observation (see Section 4). These modules are notified when an event in a subset of reasoning events occurs. Upon these occurrences additional BDI reasoning events are dispatched in the surrounding agent. Realizations of these modules contain the configuration of (1) the events that are to be added by certain observations and (2) the execution context that permits the addition of the event. This configuration can be described as a set (K) of tuples $\langle te_s, te_d, \lambda, \kappa \rangle$:

- the element te_s is a triggering event that is to be observed by the capability. The set of observed events is given by the set S ($te_s \in S$), which is a subset of the available reasoning events ($S \in E$).
- the element te_d is the corresponding event that is to be added to the agent reasoner when te_s is observed. The set of actuated events (D) is also a subset of the available reasoning events ($te_d \in D, D \in E$).
- the element λ denotes the logical location of event additions. Events (te_d) can be placed in the currently active intention (i) or in a new intention ($\top; \lambda \in \{i, \top\}$).
- the optional element κ is a boolean expression that defines when the event te_d is applicable to be added to the agent configuration. This expression takes into account the current agent state (ag, C, M, T) and denotes the context that permits the introduction of the additional event (te_d). The expressiveness of these statements depends on the utilized agent platform.

Three functions are introduced, which access this configuration, to simplify the semantics of agent state modifications. An auxiliary mapping function $m(x)$ is assumed that maps triggering events to corresponding events ($m : S \longrightarrow D$). According to the configurations in K this function returns the event(s) that are to be introduced ($m(te_s) = te_d$). The function $l : S, D \longrightarrow \lambda$ returns the target intention (λ) for an event mapping (S, D), i.e. two corresponding events (te_s, te_d). λ can have two values and indicates either that the event is to be added to the current intention (λ_c) or that the event is to be included in a new intention (λ_n). In the following, we also assume the availability of a platform specific function $eval(te_s, te_d)$ that extracts the agent execution context, looks-up the corresponding condition statement (κ) and evaluates the statement, i.e. returns a boolean ($eval : S, D \longrightarrow \{true, false\}$). The functions m and l are prescribed by the agent programmer. The given mapping defines the set of observed events and their counterparts, which are to be induced. For each of the latter events, its is also specified whether it is to be placed in the current intention or in a new one. New intentions initially contain only the triggering event and this placement initializes another concurrent course of actionfor the agent.

Implementations of CECs are aware of the mapping function and dispatch the corresponding event in the surrounding agent. Afterwards the agent execution continues unaltered. Therefore, only the event selection rule *SelEv* is supplemented with another rule (*SelEvCEC*) that defines the the contribution of a CEC when events are selected that are in the set of observed events S. If $te \notin S$, the unaltered selection rules (SelEv1,2) is used (cf. Section 4.1.1). The inserting of the event ($te_d \in D$) that corresponds the observed event $m(te)$ into the agent circumstance (C'_E) is guarded by the annotated condition (κ) that is evaluated for the currently handled event mapping ($eval(te, m(te))$). Events are inserted into the intention that is indicated by the function $l(te_s, te_d)$. The original event (te), which was selected for processing ($S_E(C_E) = \langle te, i\rangle$), is added to the temporal structure (T'_ε) and subsequently processed by the reasoning cycle. This rule does not enforce immediate event processing. The additional event (te_d) is added to the events that will be subsequently processed by the agent but the operation of the interpreter decides how agent execution proceeds.

$$\textbf{SelEvCEC} \quad \frac{S_E(C_E) = \langle te, i\rangle \qquad (te \in S)}{\langle ag, C, M, T, SelEv\rangle \longrightarrow \langle ag, C', M, T', RelPl\rangle} \tag{3}$$

where:

$$C'_E = \begin{cases} C_E \cup \{\langle m(te), l(te, m(te))\rangle\} \backslash \{\langle te, i\rangle\} & if\, eval(te, m(te)) = true \\ C_E \backslash \{\langle te, i\rangle\} & otherwise \end{cases} \qquad T'_\varepsilon = \langle te, i\rangle$$

4.2 Prototype Realization

The described mechanism has been implemented using the *Jadex* reasoning engine[3], that provides an execution environment for BDI-style agents on top of arbitrary distributed systems middleware [3]. The Jadex platform allows implementations of a certain interface (jadex.run-time.ISystemEventListener), to be registered at individual agents. The *Jadex Introspector* and *Tracer* development tools, accompanying the Jadex distribution, use this mechanism to observe agent reasoning. Implementations of this interface can be registered for a specified set of reasoning events (defined in jadex.runtime.Systemevent) and an API allows to modify the BDI facilities of the surrounding agent via an API.

4.3 Structuring Agents with Active, Context-Aware Modules

In terms of *comprehensibility* and *reusability*, it is advisable to cluster distinguishable functionalities in modules. Capabilities contain libraries of BDI elements that can be referenced to reuse functionality. Besides, the goal-oriented and event-based processing of the reactive planning mechanism, can as well be exploited to handle functionalities that would be commonly understood as crosscutting concerns. For example, message based communication are encapsulated in [10] in an *interaction* aspect while the modularization of roles in a negotiation protocol is the prime example for BDI-based capabilities [6] (cf. section 2.3). The proposed extension to the capability concept allows to automate the references to elements that are contained in capabilities. This facilitates information-hiding, as both the agent elements and the information when these are to be activated/modified are contained in a single entity.

In addition, this approach facilitates the provision of *additional*, quasi *contributive*, processing of BDI reasoning events. The original handling of events leads to a series of subsequent events. For example the achievement of a goal may involve the successive achievement of subgoals. Besides this sequential execution path, which is controlled by the agent reasoner, the described mechanism allows to declare additional activities should be activated as well. These are caused by the activation of reasoning events (see equation 3). The authority of the reasoning mechanism, e.g. a BDI interpreter, is untouched and the contributive events integrate in the currently active or a in a concurrently executed intention. An example usage is the embedding of monitoring routines that decide the significance of agent state changes and communicate these to remote observers [30] as well as the embedding of assertion validations [28].

The here discussed *activation* of agent modules reveals another perspective on the modularization of agents. Besides functional clusters (see Section 2.3), agents can be structured as composites of context-aware actors that decide locally when to provide the contained functionalities. Despite the outlined benefits, the presented mechanism reduces the *traceability* of the agent actions. it makes it more difficult to trace the causes of specific agent actions. Since the additional events are handed over to the agent reasoner for subsequent processing, arbitrary agent events can be selected for inclusion. Subsequently, all events are similarly processed. Therefore, coefficient activities can be arbitrarily selected. However, the performance of the agent functionality is reduced by the coefficient inclusion of exhaustive processings.

4.4 Construction of Coordination Endpoints

A requirement for the realization of the Decomas architecture is the ability to associate Endpoints with agents and enable these endpoints to observe, reason about, and influence the agent execution (see Section 3). The discussed coefficiency mechanism allows to construct Coordination Endpoints as agent modules. Conceptually, this approach is attractive since the agent coordination is separated is represented as

[3]http://jadex.informatik.uni-hamburg.de

an orthogonal aspect. The uncommunicative overhead, imposed by remote endpoints is avoided. Using coefficient agent modules, a generic Endpoint model has been conceived that abstracts from the utilized agent platform.

This structure is illustrated in Figure 4. A *Coordination Endpoint* is composed of three (sub-) modules. First, a *Communication* module contains the abilities to *publish* and *perceive* events that are significant for the inter-agent coordination. These modules exchange specific data elements (*Coordination Information*) and realize the Medium-specific information propagation. Endpoints contain also two interpreter elements. The *Coordination Information Interpreter* is responsible to process the perceived information and decides, based on a declarative configuration of the enacted coordination process [33], the appropriate modifications of the host agent. This is a conventional agent module except that it affects the injection of modifications in the surrounding agent (see Section 4.1.2). The *Agent State Interpreter* is a coefficient module that registers for the observation of the surrounding host agent. The events of interest are given by a declarative model of the desired inter-agent coordination [33] that also contains the conditions and constraints that control the publication of agent-internal events and data. Using *coefficiency*, the observation and affection of publications is realized as an autonomous background process inside agents.

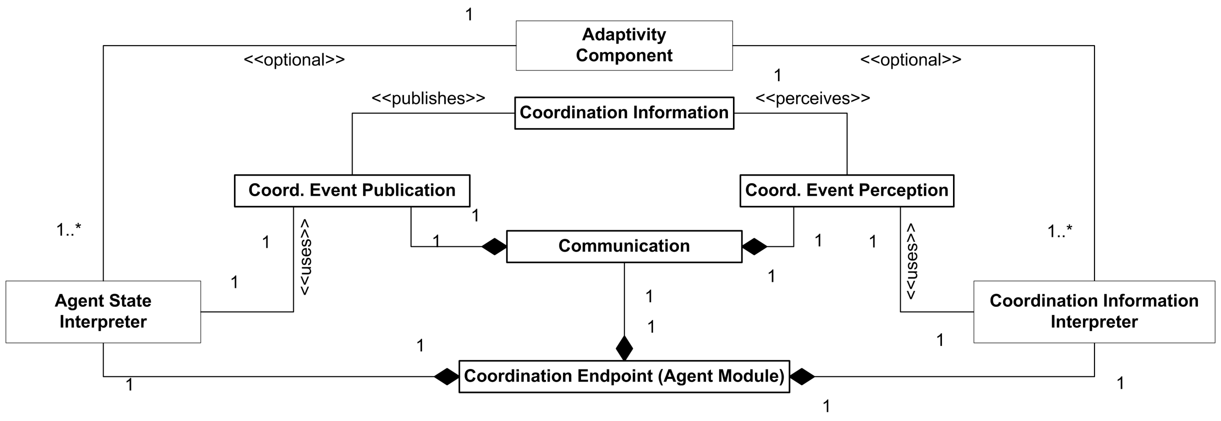


Figure 4: Excerpt from [32] that illustrates the structure of Coordination Endpoints.

5 Shoaling Glassfishes: Decentralized (Web) Service Management

Distributed software systems imply an administrative overhead that originates in the manual maintenance of computational infrastructure, e.g. the provision of server installation and server deployments in *Service Oriented Architectures*. The reduction of this overhead in dynamic environments, where request loads and resource usages fluctuate, is a prominent research topic [14]. Often it is necessary to augment existing middlewares with adaptive mechanisms [11].

Here, we outline the development of a *decentralized*, *agent-based* management framework for the maintenance of distributed software infrastructures. Figure 5 illustrates the conceptual architecture. *Services* are deployed on application server instances (*App. Server*). These servers reside on different *Server-Clusters*, i.e. are distributed in different computing / data centers. Service endpoints and application servers are managed by remote software agents (*Agent-based Management*). These agents monitor and manipulate the configurations of services and servers. The management by agents is realized with the SUN *Appserver Management EXtensions*[4] (AMX), a generic API to control J2EE Application Server configurations. Managers of application servers are bound to specific installations, while service endpoints are free to reallocate to different servers and change their service offer. Agents are equipped with plans for the deployment and undeployment of services. Unknown of service configurations can be fetched from remotely accessible *repositories*. Service consumers (cf. figure 5, top-left) invoke web services. The dynamics of service deployments and server utilizations are hidden by *Service Brokers*. These maintain local registries of the physical locations of service deployments. Therefore, clients can retrieve the addresses of the current deployments from the static locations of the Brokers. In addition, the Brokers load-balance the service utilizations.

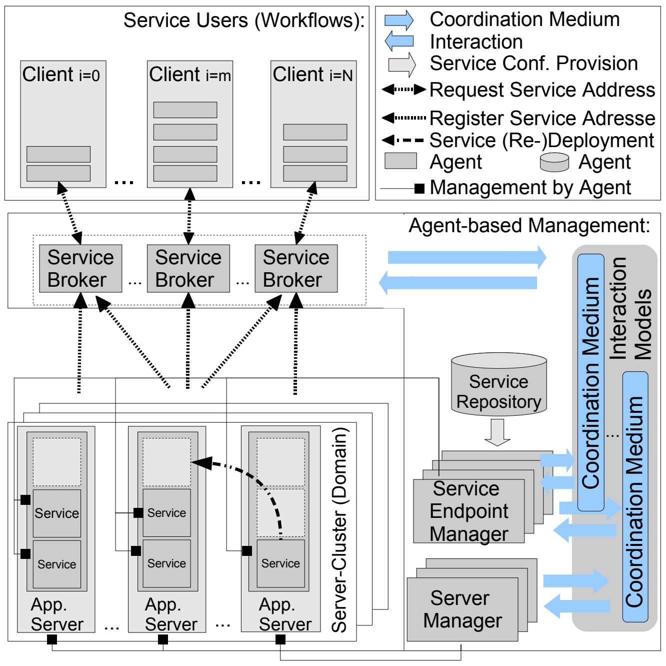


Figure 5: Decentralized, agent-based service management framework.

This prototype architecture has been realized with the *Jadex* agent framework (cf. Section 4.2). It prepares the agent-based management of service infrastructures as agents are capable to administrate services and servers, i.e. to deploy / undeploy services. The server management has been tested with the freely available *Glassfish*[5] application server. The actual coordination logic is supplemented with the systemic programming model that is discussed in Section 3. This management concerns two aspects of the dynamic deployment of (web) services. First, the allocation of physical servers is balanced to maintain averaged utilization levels. Servers are associated with preferential workload-levels. Based on the communication of available capacities, services are moved to ensure that all servers are in their preferred operational condition. Secondly, the adjustment of static service deployments is automated to meet dynamically changing service workloads. The deployments of highly-demanded services are reinforced and less-demanded services are reduced.

[4]https://glassfish.dev.java.net/javaee5/amx/index.html
[5]Version 2 (ur2-b04), https://glassfish.dev.java.net/

5.1 Server Utilization Management

The aim of the utilization management is to maintain a preferential number of service deployments on servers. The rational is, that servers are dimensioned for a specific utilization. Therefore, the maintenance of several underutilized servers is a waste of resources, e.g. energy [14], when the services could be handled by a single server that is properly utilized. The decentralized management is based on the publication of capacities by underutilized servers and services are attracted to servers that are almost well utilized.

The dynamics of the adaptive placement of services is illustrated in Figure 6 (I). The variable *Underloaded* describes the number of servers that are below their intended utilization. This utilization is maintained by a balancing feedback loop *(-)*. A Coordination Endpoint determines whether the server is underutilized or not (a logical condition, defined on the set of agent beliefs). The Endpoints within underloaded servers publish the availability of capacities (β). These publications propagate in a Coordination Medium to the Coordination Endpoints in Service Endpoint agents (*Moveable*). These decide whether to change the service deployment, i.e. move to another server, or not. Movements affect the system-wide rate of (re-)deploying agents (*Server Change*) and consequently influences the number of (re-)deployed services. Since these deployment actions were caused by the availability of resources, the number of underloaded servers decreases.

Figure 6 (II) shows simulation results for the described coordinating process. Two application servers are initialized with slightly different workloads. Servers are configured to host up to 5 services. The availability of server capacity gradually spreads in the system and services are (re-)deployed. Figure 6 (A) denotes the movement of services that is carried out be their un- and (re-)deployment.

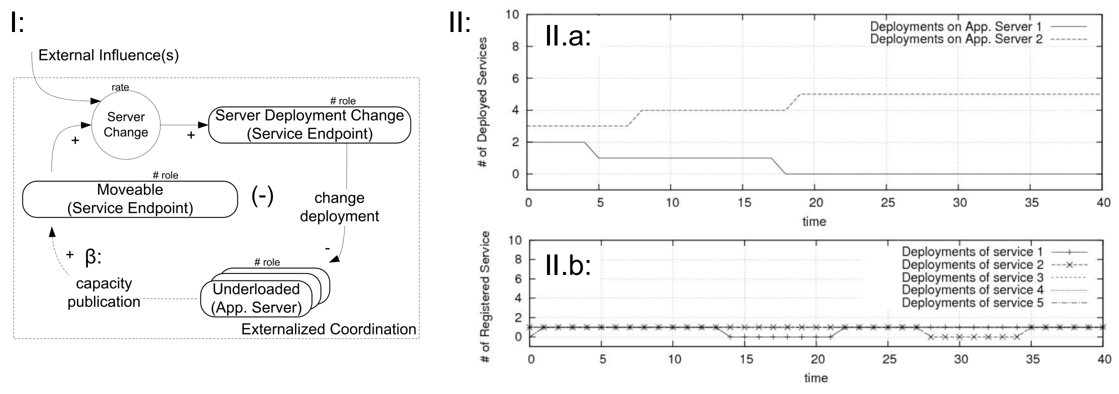


Figure 6: Server utilization management: The dynamics of the management (I) and a simulation snapshot (II). The simulation results relate the deployments per server (II.a) to all service registrations (II.b).

5.2 Balancing Service Deployments

The adaptive reinforcement of service deployments, balanced with fluctuating request loads, is based on the publications of demand changes by Broker agents. Using the coordination architecture, the measurement and interpretation of demand changes is encapsulated in Endpoint modules. These publish significant changes and upon the reception of these information, the Endpoints of Service agents decide whether to adjust the local deployment or not.

The dynamics of this adaptive process is illustrated in Figure 7 (I) *Requesters* increase, with a fluctuating rate, the number of *Service Requests* that the system has to work off. The amount of requests causally influences the measured *Service Demand*. The *Allocated* variable denotes the number of agents that offer services. Service Brokers are supplemented with the ability to publish (α) significant changes in request workloads (*Service Reinforcement*). The publications affect that agents consider an adjustment of their current service offer (*Changing Service Allocation*). The additional behaviors, required to participate in this process, are implemented in Coordination Endpoints. Within *Service Brokers*, these decide the significance of workload changes and in *Service Endpoints*, these decide whether to change deployments or not.

Figure 7 (II) shows simulation results for 10 virtual application servers (domains) that run on a Glassfish installation. Each of them is configured to host a maximum of 5 web service implementations. An additional constraint is that every service type is only deployed once on every domain. Initialized with an arbitrary configuration of service deployments, the system is exposed to a sudden workload of service type 1. The publication of this demand change enforces that Service Endpoints locally adjust and switch their deployments. Details on the integration of this coordination model and the declaration of agent-internal data to be communicated, including code fragments, can be found in [34, 33].

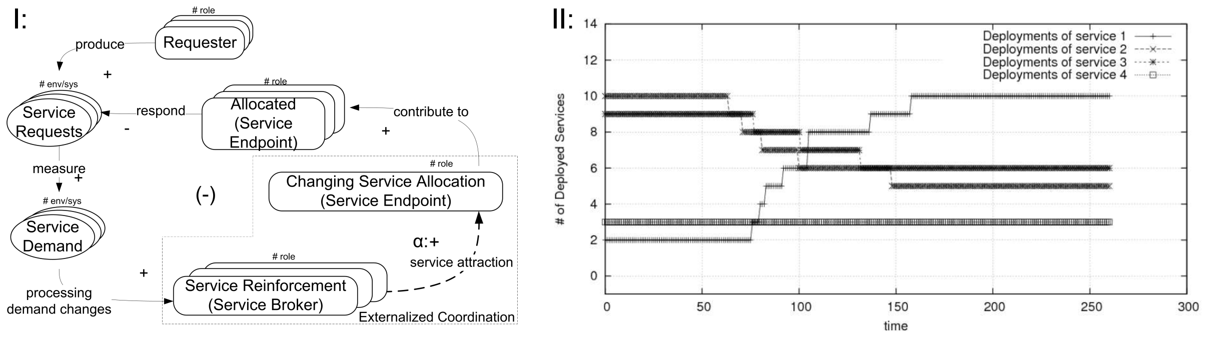


Figure 7: J2EE (Web-)Service management: Balancing Service Workload.

6 Conclusions

In this paper, we presented an architecture for the integration of decentralized, self-organizing coordination in MAS. This architecture provides a middleware layer that contains coordination mechanisms and automates their invocation. A key design concern is that the architecture can be integrated in established agent architectures and that coordination can be supplemented to functional MAS. This allows that self-organization can be supplemented to conventionally developed MAS. The accomplishment of this design objective requires that invocations can be supplemented without affecting the structure of the original, i.e. not self-organizing, MAS. This has been approached with the concept of *activated* agent modules. Their generic structure and operating principle is discussed. Their implementation is outlined and formalized for a particular agent architecture. The utilization of the coordination framework is exemplified for the management of service oriented computing infrastructures. Future work comprises the programming language techniques that are used to control the execution of Media and Endpoint elements. Ongoing work concerns the revision of a declarative configuration approach [33] to integrate the prescriptions of self-organizing processes in MAS development frameworks.

Acknowledgment

We cordially thank Rafael H. Bordini[6] from the *Institute of Informatics* at the Federal University of Rio Grande do Sul for discussing and reviewing the formalization of the endpoint operation principle that is discussed in Section 4.1. We also thank the *Distributed Systems and Information Systems* (VSIS) group at Hamburg University, particularly W. Lamersdorf, L. Braubach, A. Pokahr, and A. Vilenica for helpful discussion. The SodekoVS-project is funded by the *Deutsche Forschungsgemeinschaft* (DFG)[7].

References

[1] E. Bonabeau, M. Dorigo & G. Theraulaz (1999): *Swarm Intelligence: From Natural to Artificial Systems*. Santa Fe Institute Studies on the Sciences of Complexity. Oxford University Press.

[2] Rafael Bordini, Lars Braubach, Mehdi Dastani, Amal El Fallah Seghrouchni, Jorge Gomez-Sanz, Joao Leite, Gregory O'Hare, Alexander Pokahr & Alessandro Ricci (2006): *A Survey of Programming Languages and Platforms for Multi-Agent Systems*. In: Informatica 30, pp. 33–44.

[3] L. Braubach, A. Pokahr & W. Lamersdorf (2005): *Jadex: A BDI Agent System Combining Middleware and Reasoning*. In: Software Agent-Based Applications, Platforms and Development Kits, Birkhäuser.

[4] Lars Braubach, Alexander Pokahr & Winfried Lamersdorf (2005): *Extending the Capability Concept for Flexible BDI Agent Modularization*. In: Proc. of PROMAS-2005.

[5] Yuriy Brun, Giovanna Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè & Mary Shaw (2009): *Software Engineering for Self-Adaptive Systems*, chapter Engineering Self-Adaptive Systems through Feedback Loops, pp. 48–70. Springer-Verlag, Berlin, Heidelberg.

[6] Paolo Busetta, Nicholas Howden, Ralph Rönnquist & Andrew Hodgson (2000): *Structuring BDI Agents in Functional Clusters*. In: ATAL '99, Springer, pp. 277–289.

[7] Mehdi Dastani, M. Birna van Riemsdijk, Joris Hulstijn, Frank Dignum & John-Jules Ch. Meyer (2005): *Enacting and Deacting Roles in Agent Programming*. In: Lecture Notes in Computer Science, 3382.

[8] T. DeWolf & T. Holvoet (2007): *Decentralised Coordination Mechanisms as Design Patterns for Self-Organising Emergent Systems*. In: Engineering Self-Organising Systems, 4335/2007, pp. 28–49.

[9] Alessandro Garcia, Uir Kulesza & Carlos Lucena (2005): *Aspectizing Multi-agent Systems: From Architecture to Implementation*. In: Lecture Notes in Computer Science, 3390, pp. 121 – 143.

[10] Alessandro F. Garcia, Carlos J. P. de Lucena & Donald D. Cowan (2004): *Agents in object-oriented software engineering*. Softw. Pract. Exper. 34(5), pp. 489–521.

[11] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl & P. Steenkiste (2004): *Rainbow: architecture-based self-adaptation with reusable infrastructure*. Computer 37(10), pp. 46–54.

[12] David Gelernter & Nicholas Carriero (1992): *Coordination languages and their significance*. Commun. ACM 35(2), pp. 97–107.

[13] Koen Hindriks (2008): *Modules as Policy-Based Intentions: Modular Agent Programming in GOAL*. In: Programming Multi-Agent Systems, 4908/2008, Springer Berlin / Heidelberg.

[14] Rajarshi Das Jeffrey, O. Kephart, Charles Lefurgy, Gerald Tesauro, David W. Levine & Hoi Chan (2008): *Autonomic Multi-Agent management of Power and Performance in Data Centers*. In: Proc. of the 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), pp. 107–114.

[15] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin (1997): *Aspect-Oriented Programming*. In: Proc. of ECOOP, Springer.

[6] R.Bordini@inf.ufrgs.br

[7] http://www.dfg.de

[16] Cidiane Lobato, Alessandro Garcia, Alexander Romanovsky, Cludio Sant'Anna, Uir Kulesza & Carlos Lucena (2004): *Mobility as an Aspect: The AspectM Framework*. In: *Proceedings of the 1st Brazilian Workshop on Aspect-Oriented Software Development WASP04*.

[17] D. L. Parnas (1972): *On the criteria to be used in decomposing systems into modules*. *Commun. ACM* 15(12), pp. 1053–1058.

[18] Gordon D. Plotkin (2004): *The origins of structural operational semantics*. *Journal of Logic and Algebraic Programming* 60-61, pp. 3–15. Available at `http://dx.doi.org/10.1016/j.jlap.2004.03.009`.

[19] Mikhail Prokopenko (2008): *Advances in Applied Self–organizing Systems*, chapter Design vs. Self–organization, pp. 3–17. Springer London.

[20] A. S. Rao & M. P. Georgeff (1995): *BDI-agents: from theory to practice*. In: *Proceedings of the First Int. Conference on Multiagent Systems*. Available at `citeseer.ist.psu.edu/rao95bdi.html`.

[21] Anand S. Rao (1996): *AgentSpeak(L): BDI agents speak out in a logical computable language*. In: *Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world*, pp. 42–55.

[22] M. Birna van Riemsdijk, Mehdi Dastani, John-Jules Meyer & Frank de Boer (2006): *Goal-Oriented Modularity in Agent Programming*. In: *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM Press.

[23] Mazeiar Salehie & Ladan Tahvildari (2009): *Self-adaptive software: Landscape and research challenges*. *ACM Trans. Auton. Adapt. Syst.* 4(2), pp. 1–42.

[24] Linda M. Seiter, Daniel W. Palmer & Marc Kirschenbaum (2006): *An aspect-oriented approach for modeling self-organizing emergent structures*. In: *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*, ACM Press, New York, NY, USA, pp. 59–66.

[25] G. D. M. Serugendo, M. P. Gleizes & A. Karageorgos (2006): *Self-Organisation and Emergence in MAS: An Overview*. In: *Informatica*, 30, pp. 45–54.

[26] G. Di Marzo Serugendo & J. Fitzgerald (2009): *Designing and Controlling Trustworthy Self-Organising Systems*. *Perada Magazine* .

[27] Amit Shabtay, Zinovi Rabinovich & Jeffrey S. Rosenschein (2006): *Behaviosites: a novel paradigm for affecting Distributed Behavior*. In: *Proceedings of ESOA'06*, pp. 23–39.

[28] Jan Sudeikat, Lars Braubach, Alexander Pokahr, Winfried Lamersdorf & Wolfgang Renz (2006): *Validation of BDI Agents*. In: *Programming Multi-Agent Systems (ProMAS 2006)*, number 4411 in LNAI, pp. 185–200.

[29] Jan Sudeikat, Lars Braubach, Alexander Pokahr, Wolfgang Renz & Winfried Lamersdorf (2009): *Systematically Engineering SelfOrganizing Systems: The SodekoVS Approach*. *EASST* 17. ISSN 1863-2122.

[30] Jan Sudeikat & Wolfgang Renz (2007): *Monitoring Group Behavior in Goal–Directed Agents using Co–Efficient Plan Observation*. In: *Agent-Oriented Software Engineering VII*, 4405/2007, pp. 174–189.

[31] Jan Sudeikat & Wolfgang Renz (2008): *Applications of Complex Adaptive Systems*, chapter Building Complex Adaptive Systems: On Engineering Self–Organizing Multi–Agent Systems, pp. 229–256. IGI Global.

[32] Jan Sudeikat & Wolfgang Renz (2009): *DeCoMAS: An Architecture for Supplementing MAS with Systemic Models of Decentralized Agent Coordination*. In: *Proc. of the 2009 IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology*, IEEE Computer Society Press, pp. 104–107.

[33] Jan Sudeikat & Wolfgang Renz (2009): *MASDynamics: Toward Systemic Modeling of Decentralized Agent Coordination*. In: K. David & K. Geihs, editors: *Kommunikation in Verteilten Systemen*, pp. 79–90.

[34] Jan Sudeikat & Wolfgang Renz (2009): *Programming Adaptivity by Complementing Agent Function with Agent Coordination: A Systemic Programming Model and Development Methodology Integration*. *Communications of SIWN* 7, pp. 91–102. ISSN 1757-4439.

[35] Renata Vieira, Álvaro Moreira, Michael Wooldridge & Rafael H. Bordini (2007): *On the formal semantics of speech-act based communication in an agent-oriented programming language*. *J. Artif. Int. Res.* 29(1), pp. 221–267.

A Wave-like Decentralized Reconfiguration Strategy for Self-organizing Resource-Flow Systems

Jan Sudeikat[1], Jan-Philipp Steghöfer[2], Hella Seebach[2], Wolfgang Reif[2], Wolfgang Renz[1], Thomas Preisler[1], Peter Salchow[1]

[1]Multimedia Systems Laboratory (MMLab),
Faculty of Engineering and Computer Science, Hamburg University of Applied Sciences,
Berliner Tor 7, 20099 Hamburg, Germany

{jan.sudeikat, wolfgang.renz, thomas.preisler, peter.salchow}@haw-hamburg.de

[2]Institute for Software and Systems Engineering,
Augsburg University, Universitätsstrasse 6a, 86135 Augsburg, Germany

{steghoefer, seebach, reif}@informatik.uni-augsburg.de

In resource-flow systems, e.g. production lines, agents are processing resources by applying capabilities to them in a given order. Such systems profit from self-organization as they become easier to manage and more robust against failures. In this paper, we propose a decentralized coordination process that restores a system's functionality after a failure by propagating information about the error through the system until a fitting agent is found that is able to undertake the required function. The mechanism has been designed by combining a top-down design approach for self-organizing resource-flow system and a systemic modeling approach for the design of decentralized, distributed coordination mechanisms. The systematic conception of the inter-agent process is outlined and initial evaluations show the convergence to stable, i.e. fully operative configurations.

A domain in which self-organization is beneficial are production automation systems. In traditional production lines, resources are transported from one workstation to the other on conveyor belts or with similar, rigid and inflexible transportation mechanisms. While relatively easy to manage, these systems come to a complete halt whenever one of their constituent parts ceases to function. Also, if the requirements of the production lines change, the system will have to be retooled, a difficult, expensive, and time-consuming process. To add failure tolerance to such systems and to make this rigid structure more flexible, production automation systems could instead use autonomous guided vehicles (AGVs) that transport resources and robots that have several tools they can change in order to apply different capabilities as required [1]. We call this class of systems *Self-Organizing Resource-Flow Systems* with application domains in production automation and logistics. Their basic structure can be described with the Organic Design Pattern (ODP) [2]. *Resources* are processed according to a *task* by independent *agents*. Each agent has a number of *capabilities* it can apply to the resource to alter it according to the task. Agents can exchange resources with other agents as, e.g., given by the layout of a shop floor.

In this paper, we propose a decentralized, self-organizing process for the class of self-organizing resource-flow systems. This process is analyzed and modeled with the tools provided by the SodekoVS[1] project [3]. Agents change their local configurations until the system in its entirety has restored a stable state. During reconfiguration, parts of the system that are not affected by the process or have already been reconfigured are still able to resume their normal work.

The utilization of self-organization principles in the development of distributed software systems is considered in the research project "Selbstorganisation durch Dezentrale Koordination in Verteilten

[1]Acknowledgments: This research is partly sponsored by the German research foundation (DFG) in the project SodekoVS and in the DFG special priority program "Organic Computing" (SPP 1183) in the project SAVE ORCA.

Systemen" (SodekoVS) [3]. The supplementation of self-organizing features to software systems is based on the externalization of coordination models [4]. A coordination middleware serves as a reference model for the integration of decentralized self-organizing processes in MAS. It contains two types of functional elements for the encapsulation of these aspects. First, agents are connected with each other via so-called *Coordination Media*. These are communication infrastructures that allow to encapsulate specific interaction modes. Secondly, the utilization of these media is separated from the agent-logic by associating agents with *Coordination Endpoints*. They initiate and participate in medium-mediated interactions on behalf of the associated agents and effectuate modifications on agents when appropriate. This middleware separates and automates the activities, i.e. interactions and local adjustments, that are conceptually related to coordination.

This reference architecture has been used to realize a completely decentralized reconfiguration approach. It is based on the idea that a wave of role re-allocation runs through the system in order to re-establish the resource-flow. Assuming that each agent is capable to exhibit a set of capabilities, a correct resource flow can be (re-)established by the appropriate swapping of roles. Failing agents adopt actable roles and in return other agents help out by providing the unactable roles. Waves of reallocations originate from the failing agents as these send requests for assistance along the resource flow. Recipients of these request decide locally whether they are capable and willing to swap roles. Depending on the system configuration, it may be the case that a single swap of roles does not reestablish the correct sequence of activities, thus *transitive* changes of roles are required. Prior to the detailed design and embedding of this decentralized process, we anticipated the resulting system dynamics. The decentralized reconfiguration is transfered to a stochastic π-calculus model that can be simulated. Then the decentralized reconfiguration strategy has been realized on top of an agent-based simulation model of production lines. This implementation makes use of the freely available *Jadex* agent framework [3]. A realization of the *Coordination Middleware* for this agent platform is utilized [4]. Evaluations show the quick convergence to stable states and the reconfigurations only affect system partitions.

The outlined algorithm works by exchanging responsibilities with neighboring agents and by propagating change requests until all of them could be satisfied. Thus, the reconfiguration propagates through the system like a wave. An interesting aspect of the proposed mechanism is that configuration changes are preferred to happen in the neighborhood of the deficient agents. By keeping the reconfiguration local, other parts of the system are not impaired by a failure and can continue to run normally. Future work includes a more detailed elaboration of the combination of top-down design methodologies as promoted with the ODP and bottom-up design of coordination methods as proposed in SodekoVS. This will also include a comparison of their respective advantages and problems.

References

[1] A. Hoffmann, F. Nafz, H. Seebach, A. Schierl & W. Reif (2010): *Developing Self-organizing Robotic Cells using Organic Computing Principles*. In: *Workshop on Bio-Inspired Self-Organizing Robotic Systems, Proceedings of the International Conference on Robotics and Automation 2010*.

[2] H. Seebach, F. Ortmeier & W. Reif (2007): *Design and Construction of Organic Computing Systems*. *IEEE Congress on Evolutionary Computation, 2007*, pp. 4215–4221.

[3] J. Sudeikat, L. Braubach, A. Pokahr, W. Renz & W. Lamersdorf (2009): *Systematically Engineering SelfOrganizing Systems: The SodekoVS Approach*. *Electronic Communications of the EASST* 17. ISSN 1863-2122.

[4] J. Sudeikat & W. Renz (2009): *Programming Adaptivity by Complementing Agent Function with Agent Coordination: A Systemic Programming Model and Development Methodology Integration*. *Communications of SIWN* 7, pp. 91–102. ISSN 1757-4439.

A 90% RESTful Group Communication Service [*]

Tadeusz Kobus
Poznań University of Technology

Paweł T. Wojciechowski
Poznań University of Technology

In this paper we introduce a 90% RESTful group communication service that we have developed for resilient Web applications. Our system is based on Spread—a popular group communication toolkit which delivers many useful programming abstractions, such as various reliable ordered broadcasts; they can be used, e.g. for implementing systems tolerant to server crashes. Contrary to Spread and many other such systems available as libraries of programming languages, we represent group communication abstractions as resources on the Web, addressed by URIs. To our best knowledge, this is the first approach to engineering group communication systems in this way.

The *Web* can provide a common, language-independent platform for interoperable resilient services that work together to create seamless and robust systems. *Service resilience*, defined as the continued availability of a service despite failures and other negative changes in its environment, is vital in the *Service-Oriented Architecture (SOA)*. We must ensure that each service is highly available regardless of unpredictable conditions, such as sudden and significant degradation of network latency or failure of dependant services. In this paper, we sketch our work on *group communication* service which can be used for implementing resilient Web services, based on REST [1]; see the technical report [3] for a full description.

A typical way of increasing service resilience is to replicate it. *Service replication* means deployment of a service on several server machines, each of which may fail independently, and coordination of client interactions with service replicas. A general model of such replication is called *replicated state machine* [5]. The key abstractions required to implement this model are provided by *group communication systems*. They provide various primitives for: a) detection of malfunctioning/crashed processes, b) reliable point-to-point transfer of messages, c) formation of processes into groups, the structure of which can change at runtime, and d) reliable message multicasting with a wide range of guarantees concerning delivery of messages to group members (e.g. causally- , fifo- and totally-ordered message delivery). Notably, the overlay protocols for reliable multicasting do not depend on any central server, so that there is no single point of failure. For this, *distributed agreement* protocols must be used.

For the past 20+ years, many group communication systems have been implemented (see [3] for references). Unfortunately, various group communication systems usually have quite different application programming interfaces, which are non-standard, complex and language/platform dependent. Moreover, many of such systems are monolithic, i.e. it is not possible to replace their protocols or add new features. Using these systems to implement SOA services makes the code of services not easily reusable nor manageable (adaptable), which is a counterexample to the Service-Oriented Architecture.

We propose an approach to designing an API of a group communication system, which is based on the *REpresentational State Transfer (REST)* [1]. Typically REST uses HTTP and its methods GET, PUT, POST, and DELETE. This makes it easy to describe one RESTful Web service call to another Web service, e.g. a replicated service call to a group communication service. We need only supply a verb, a URI and (optionally) a few headers containing the message payload. Thus, REST gives us a

[*]This work has been partially supported by the Polish Ministry of Science and Higher Education within the European Regional Development Fund, Grant No. POIG.01.03.01-00-008/08.

uniform, simple way of using group communication abstractions by Web applications, fulfilling the SOA requirements, such as language/platform independence and easy software integration. Moreover, the use of HTTP usually enables us to communicate with servers behind fire-walls.

However, we had to solve some problems to realize this approach, mostly related to the REST characteristics and the constraints imposed on HTTP. For example, the client would not be able to change state based on the responses of intermediary service calls; also, we had to provide means for the client to communicate both synchronously and asynchronously with the group communication service using purely the HTTP methods; and we needed to match error codes of the HTTP protocol to the incorrect behaviour of a group communication system.

Various authors pointed out significant limitations of the REST architecture style. For example, Khare and Taylor [2] discussed some of the limitations and proposed several extensions of REST (collectively called *ARRESTED*). They allow to model the properties required by distributed and decentralized systems. Similarly to them, we are not bound by the rules of the original model. REST cannot model group communication well. Therefore our goal was rather to design the RESTful interface to group communication, albeit sacrificing strict conformance to the original REST model. To emphasize that group communication cannot fully conform to REST, we say that our approach is “90% RESTful”.

To illustrate our ideas, we have been developing *RESTGroups*—a group communication programming tool that can be used for developing resilient services on the Web. RESTGroups is an extension of Spread [6] with a daemon and an API based on (some % of) REST over the standard HTTP. We think that the benefits of using our tool overcome the lack of REST purity. RESTGroups functions as a front-end for Spread that is architecture- and language-independent, i.e. communicating services can be implemented with the use of a variety of programming languages and can run on different platforms.

RESTGroups provides a representation of Spread group communication services as resources, identified by URIs, with a simple but powerful API that only uses the following methods of the HTTP protocol for invoking the services: GET is used to retrieve data (for example, a message) or perform a query on a resource; the data returned from the RESTGroups service is a representation of the requested resource; POST is used to create a new resource (for example, to extend a process group with a new process or to send/broadcast a new message); the RESTGroups service may respond with data or a status indicating success or failure; PUT is used to update existing resources or data; and DELETE is used to remove a resource or data (for example, to remove a process from a process group). In some cases, the update and delete actions may be performed with POST operations as well. To our best knowledge, it is the first attempt to a RESTful group communication service; the distribution files and *javadoc* are available [4].

References

[1] Roy T. Fielding & Richard N. Taylor (2002): *Principled design of the modern Web architecture*. *ACM Transactions on Internet Technology (TOIT)* 2(2), pp. 115–150.

[2] Rohit Khare & Richard N. Taylor (2004): *Extending the Representational State Transfer (REST) Architectural Style for Decentralized Systems*. In: *Proceedings of ICSE ’04*, pp. 428–437.

[3] Tadeusz Kobus & Paweł T. Wojciechowski (2010): *A 90% RESTful Group Communication Service*. Technical Report RA-02/10, Institute of Computing Science, Poznań University of Technology.

[4] RESTGroups (2010). `http://www.cs.put.poznan.pl/pawelw/restgroups/`.

[5] Fred B. Schneider (1993): *Replication management using the state-machine approach*. In: Sape Mullender, editor: *Distributed Systems (2nd Ed.)*, ACM Press/Addison-Wesley Publishing Co., pp. 169–197.

[6] Spread Concepts LLC (2006). *The Spread Toolkit*. `http://www.spread.org/`.