Programming distributed systems with Varda

PhD of Laurent Prosperi

Advisors:
Marc Shapiro
Ahmed Bouajjani
Mesaac Makpangou

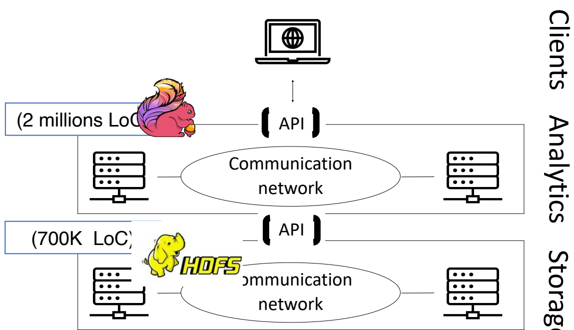
Distributed systems

Distributed systems
- Remote logical units
- Communicate through network

Intrinsic complexity

"Impossible to know what personal data is processed by what systems and when"
2021 Facebook Papers leak

Composition = re-use and assemble
- Components (e.g., processes, systems)
  - Heterogeneous
  - Often off-the-shelf (OTS)
- API interconnection



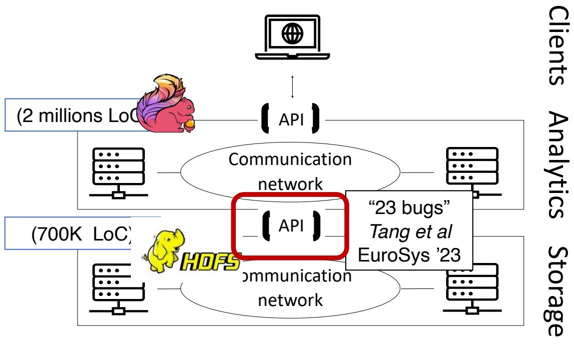
Composition limits

Interaction failures
- Hard to detect / to fix

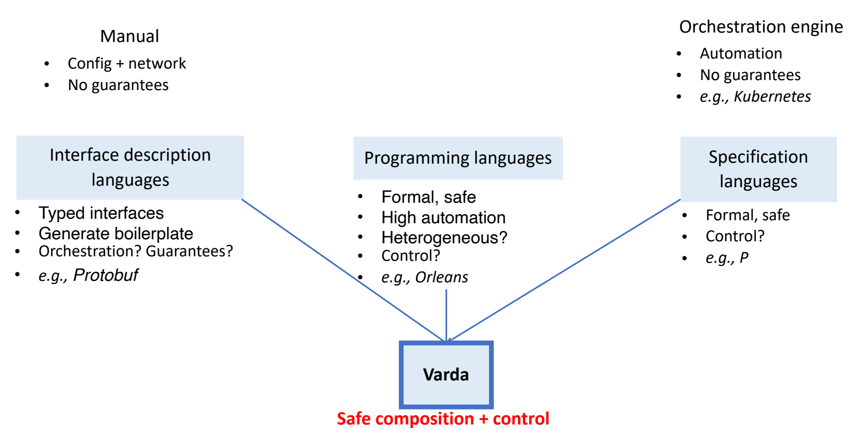
"120 *cross-system interaction* (CSI) failures"
Tang et al
EuroSys '23

Distributed systems are critical

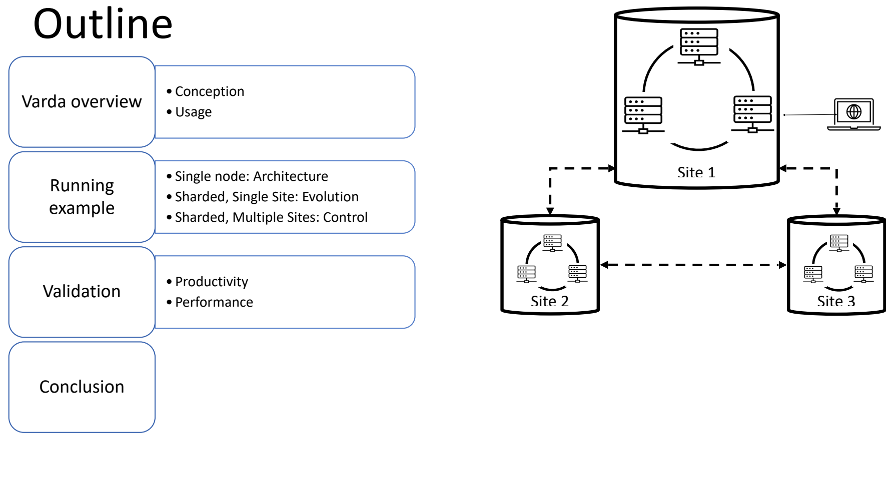
Les Echos
Orange encore touché par une panne des appels vers les numéros d'urgence
17 Janv. 2023



Approaches to composition



Outline



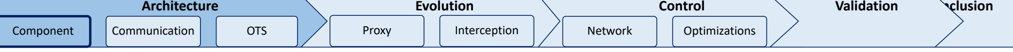
Varda overview

Developer interviews

Requirements for systems programmers

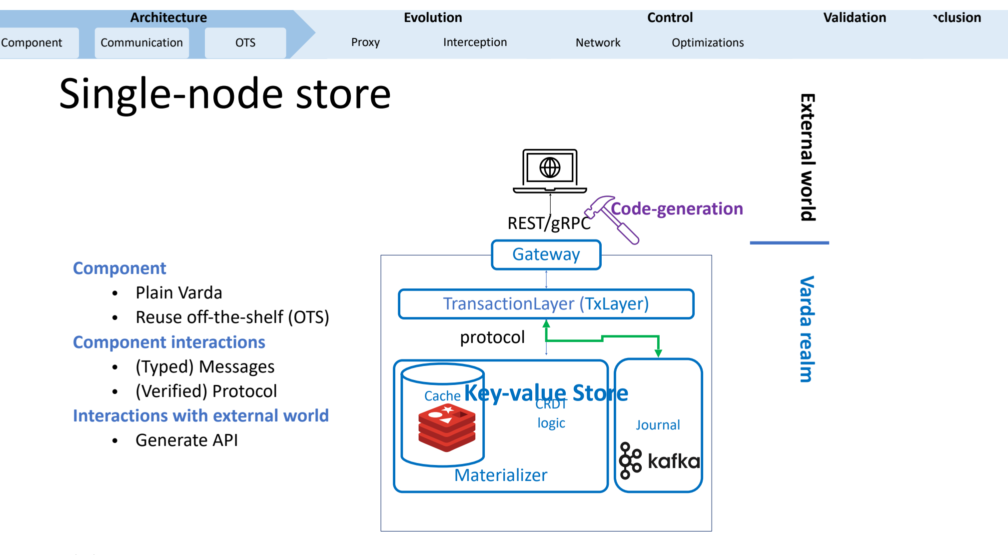
Compiler: verify, generate, optimise

- Components: off the shelf (OTS) + bespoke
- Interactions: verify, generate code
- Fine-grain control
  - OTS adaptor
  - Placement: collocate, anti-collocate
  - Inline: direct invocation
  - Auditable
- Controlled re-use & evolution
  - Interposition



Core Varda architecture

Single-node store

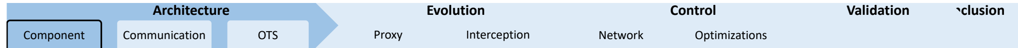


Single-node store

Component
- Plain Varda
- Reuse off-the-shelf (OTS)

Component interactions
- (Typed) Messages
- (Verified) Protocol

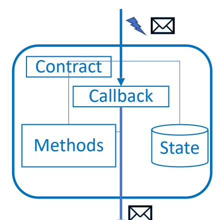
Interactions with external world
- Generate API

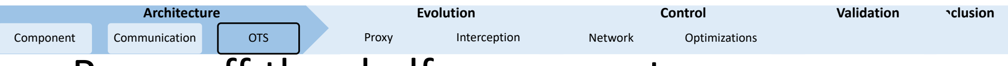


Varda component

Component = smallest distributed entity
- React to events / messages
- Perform local action / send messages
- Spawn new components
- Non-blocking

Guarantees
- Strong isolation
- Constrained behaviour
  - Contract = pre/post conditions





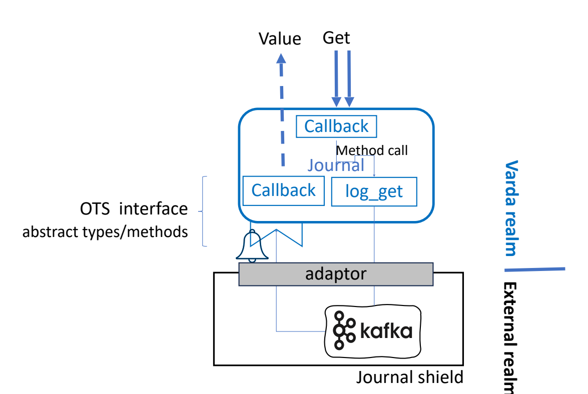
Reuse off-the-shelf components

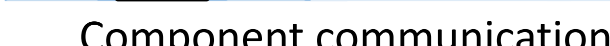
Shielding Kafka

Black box, maybe buggy
- Shield sandboxing
- Adaptors

Heterogeneous technologies
- Abstract OTS interface
- Linked at compile-time

React to asynchronous notification
- Supervision ports
  Kafka notification => Varda message



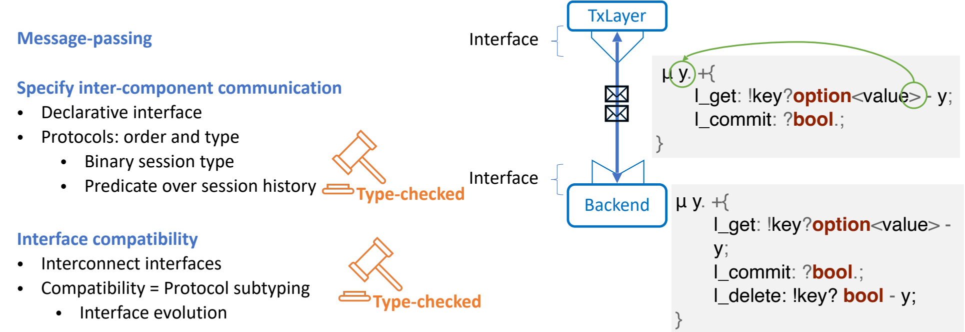


Component communication

Message-passing

Specify inter-component communication
- Declarative interface
- Protocols: order and type
  - Binary session type
  - Predicate over session history

Interface compatibility
- Interconnect interfaces
- Compatibility = Protocol subtyping
  - Interface evolution



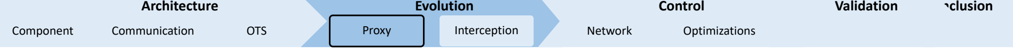
Summary

Running Varda architecture
- Components (Plain, OTS)
- Protocols + orchestration logic

Guarantees
- Isolation
- Type-checking = Protocols
- Run-time checks
  - Contracts
  - Protocol predicates

Evolving the architecture

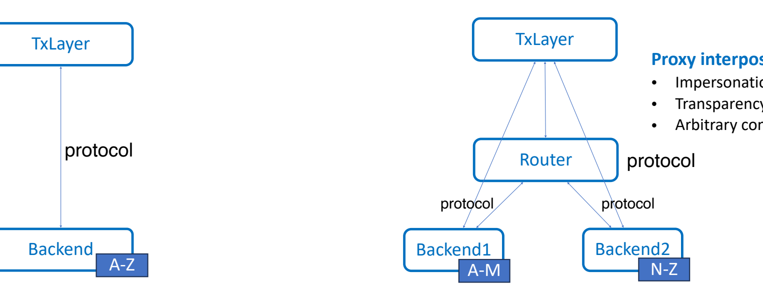
Sharded, single site



Sharding the store

Current state

Desired end state



Proxy interposition
- Impersonation
- Transparency
- Arbitrary computation

Model proxy interposition = Interception
Prosperi et al Netys'22

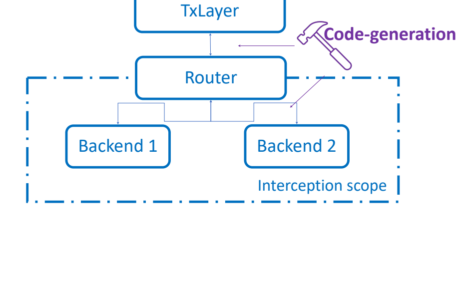
Expressing interception

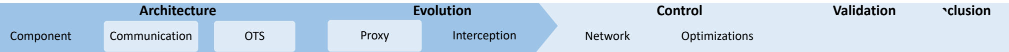
1. Proxy = Interceptor
2. Intercepted components ∈ Interception scope

```
intercept<Router> router_policy
{
  backend_1 = spawn
Backend(…);
  backend_2 = spawn
Backend(…);
```

3. …er_policy

```
activation_ref<Router> router_policy(...){
  if(this.router == none()){
    this.router = (* create one router *)
  }
  return this.router;
}
```





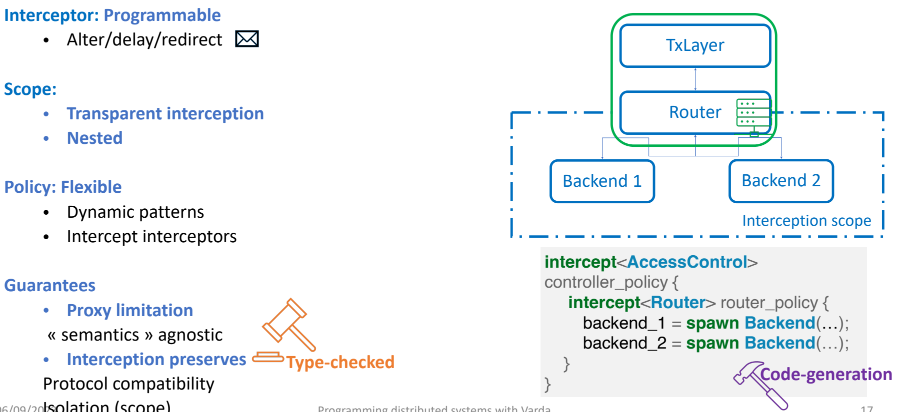
Interception properties

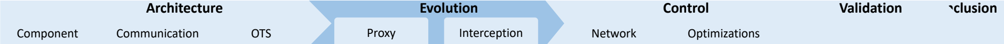
Interceptor: Programmable
- Alter/delay/redirect

Scope:
- **Transparent interception**
- **Nested**

Policy: Flexible
- Dynamic patterns
- Intercept interceptors

Guarantees
- **Proxy limitation**
  « semantics » agnostic
- **Interception preserves**
  Protocol compatibility
  Isolation (scope)

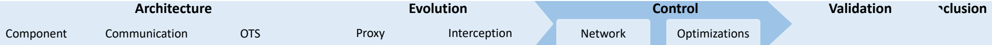




Summary

Proxy interposition = interception
- Sharding
- Replication
- Message piggy-packing
- Access control

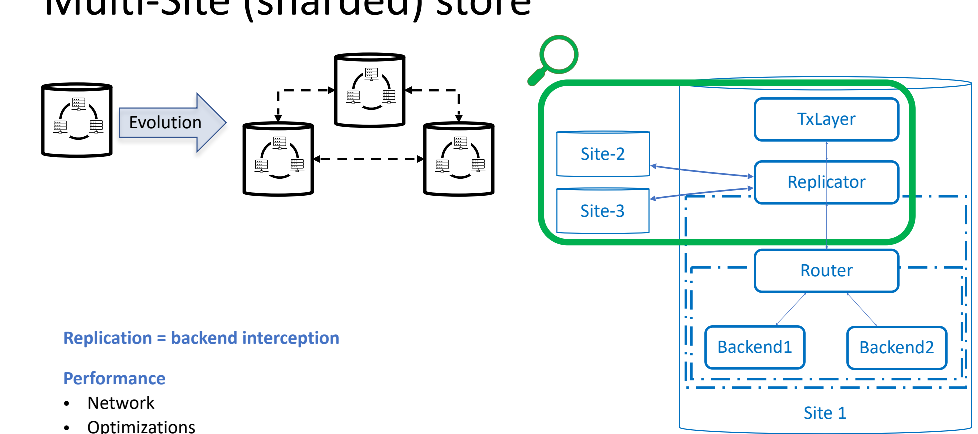
Guarantees
- Legality of interception
  (type-checking)
- Isolation



Controlling non-functional properties

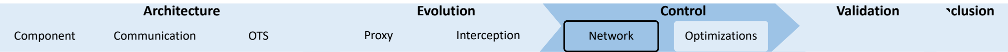
Sharding, Multi-Sites

Multi-Site (sharded) store



Replication = backend interception

Performance
- Network
- Optimizations



Network

Fine-grain control

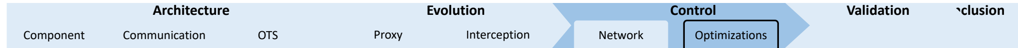
- Intra-Site: low-latency + no network partition
- Inter-Site: asynchronous + network partition

Channel = first-class object
Abstract network link

TxLayer
Raw socket
Site-2
Replicator
Site-3
Site-1
RabbitMQ

- Channel: not Varda programmable
- Channel = external library (FIFO)
- Assumptions: Point to point FIFO

```
channel<Replicator, Replicator, p_server> inter_site =
amqp_channel(
        "broker_address", "topic_name");
bind(replicator, inter_site);
```

Optimizations

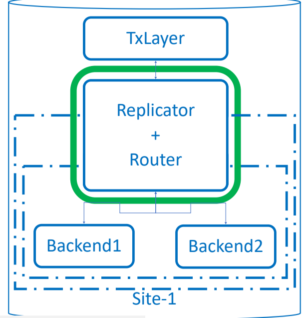
Modularization ⇒ overhead

- Components ⇒ isolation + context switching
- Interception ⇒ indirections

Varda optimizations

	Mitigate	Scope of use
Co-location	Network overhead	Generic
Local messages	Serialization overhead	Target specific
Component inlining	Context switching	Generic

- Maintains logical isolation



router = **spawn Router** (…) **in** replicator;

Summary

Control

- Network = channels
- Placement
- Supervision
- Optimizations

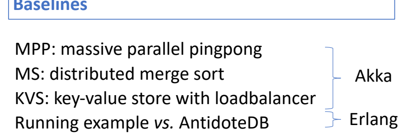
Validation

Method

Metrics

Compactness and conciseness
Lines of Code (LoC)
Performance overhead
Latency

Baselines



MPP: massive parallel pingpong
MS: distributed merge sort
KVS: key-value store with loadbalancer
Running example *vs.* AntidoteDB

Based on our Vardac compiler

Compiler	OCaml (38 KLoC)
Java dist-lib	Java11 + Akka (4 KLoC)
Tests+benchmarks	Varda/Java (20KLoC)

Productivity

Compactness

Baseline | Varda | Varda adaptor

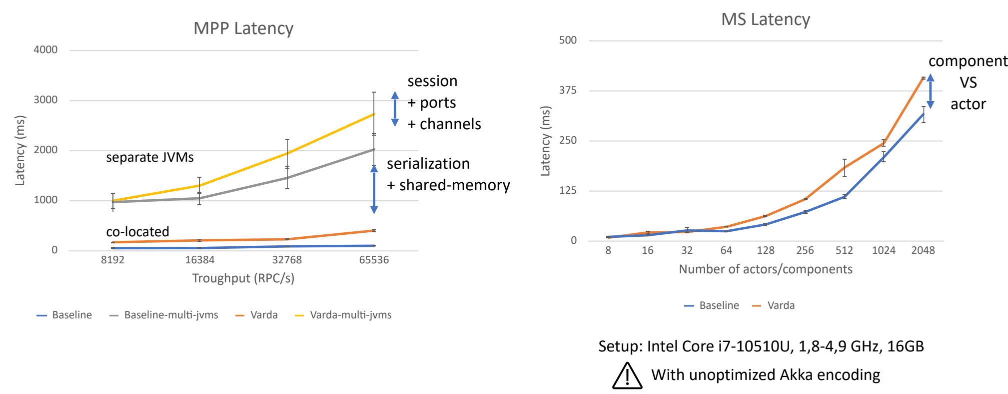
Normalised LoC (%)

100 80 60 40 20 0

MPP MS KVS AntidoteDB

	Baseline LoC	Varda LoC	Varda adaptor LoC
MPP	310	133	10
MS	338	173	48
KVS	661	181	30
AntidoteDB	4000 (13500)	956	184

Performance



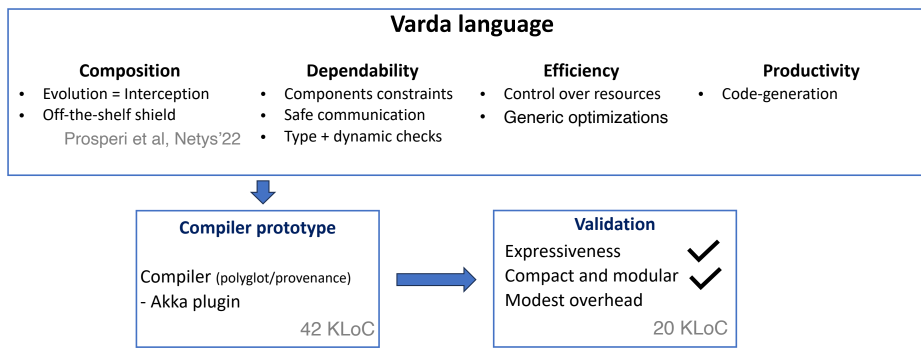
Setup: Intel Core i7-10510U, 1,8-4,9 GHz, 16GB

⚠ With unoptimized Akka encoding

Conclusion

Contributions

Requirements
- Safe composition
- End-users = system programmers
- Control over performances



Compiler prototype

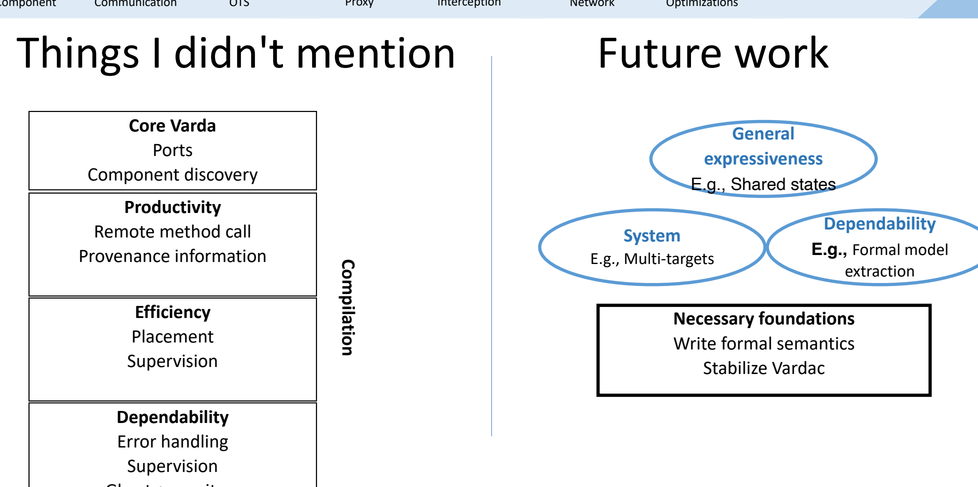
Compiler (polyglot/provenance)
- Akka plugin

42 KLoC

Validation

Expressiveness ✓
Compact and modular ✓
Modest overhead

20 KLoC



What's new

Specific problem: sound and easy composition WITH control

⇒ Varda requirements differ from SoA

Originality of the Varda language

1. Position: Architecture = intermediate representation
   - Modular architecture
     - Concise style: evolution + communication
   - Isolation + (Optional) safety constraints
   - (Optional) low-level mechanism
2. Ready for system programming
3. Major technical contributions
   - Interception
   - Component inlining
   - Adapt actor model with session
     non-blocking + safety + call back hell mitigation
   - Extend session types
     predicates over message/time bounds/session history

Why a new language

- Why a language: safety
- A new language
  - Full control on the conception/implementation
    - Interception / inlining
    - Mixing low-level details with high-level constructs (protocols)
      - network/place vs protocols
  - No good candidates
    - Independent from implementation
    - High-level languages = seek automation
    - Some good candidate but not mature enough

Influenced by

- Component-based model
- Architecture description language
  - Architecture + network links
- Type theory
  - Session types + Polymorphism + Subtyping
- Programming language
  - Actor model (execution model)
  - Contract/reactive/tierless programming
- Reflective system + aspect programming
  - Interception
- Orchestration engine + service mesh
  - Interception
  - Black boxes/sandboxing
- Specification languages
  - Varda position + code-generation
  - Additional properties

Interviews

Collaborative development platforms:
- Plateform.sh, XWiki and the DiverSE team

Edge and IoT computing:
- AdLink and Concordant

Storage and data management:
- AntidoteDB team and Scality

Blockchain:
- Nomadic Labs.

Guarantees

- **Global system view**
  - **No drift / UpToDate implementation** — Compiler
  - **(Centralized/static) system cartography** — By design
- **Out-of-the-box guarantees**
  - **Isolation** — Runtime
  - **Safe communication** — Type-checked
    - **Protocols: type and order**
    - **Topology: channel types**
  - **OTS shielding** — By design + runtime
- **(Optional) specification** — Dynamic-checked
  - **Protocol guards**
    - **Message/history predicates, time bounds**
  - **Contract + ghost + monitors**

Contract vs Protocols guard
- Centralize communication constraints
- $\perp$ components definitions

  Avoid manual component annotations
  One protocol <-> multiple components

Overhead control
- **specialized building blocks**
  - **E.g., guard vs monitor**
- Dynamic checks = Optional annotations
  - Compile time elimination

Comparison with the actor model

Same execution model (component)

- Why
  - Non-blocking + Isolation
  - Programmers remain in control
- Limitations
  - Callback hell
  - « Unsafe » communication
  - Current trend: automation

Component implementation

- Akka code-generation: component ⇒ actor
- (In general) Component: actors, containers, lambdas, …

Protocol expressiveness

Protocol

- Non-deterministic choices
- Recursive protocol
- Type checking

Predicates

- On message values
- On session history
- On time bound
- Run-time checks

```
μ y. +{
    l_get: !key{
        metadata string last_c="" l msg ->
            (* predicates *)
            key_predicate(msg) &&
            last_c < msg &&
             store_meta(last_c, msg)
        }?option<value> - y;
    l_commit: ?bool.;
}
```

```
μ y. +{
    l_get: !key{timer t l}?option<value>{v -> t<5} -
    y;
    l_commit: ?bool.;
}
```


Communication: « Callback hell »

« Callback hell »

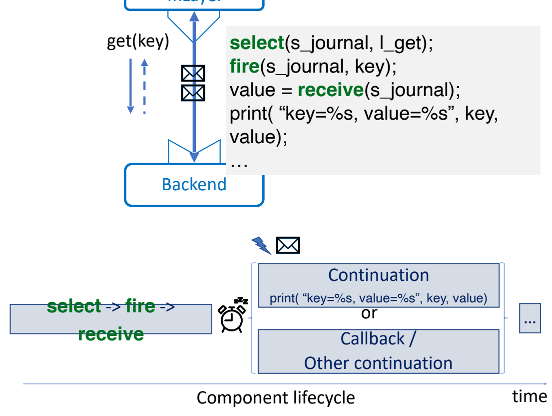
- Nested callbacks
- Hard to debug / find bottlenecks

Session: linear programming

- receive / branch / loop on stream

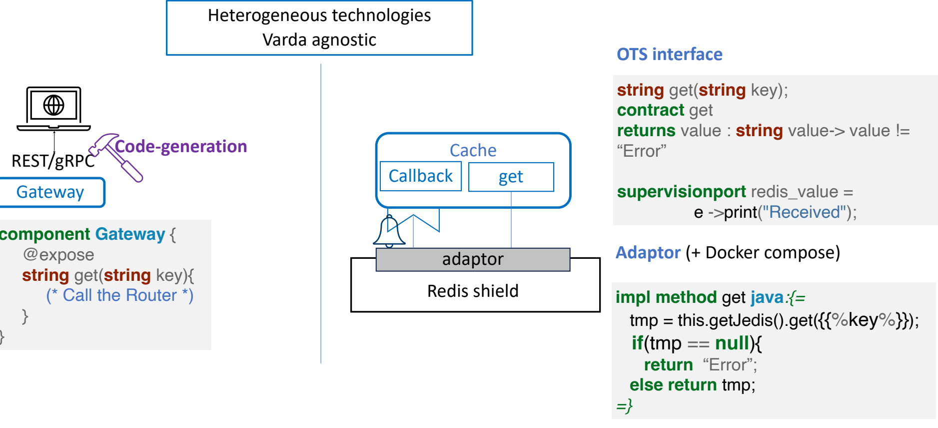
Non-blocking vs Session

- One-session = linear programming
- Inter-sessions interleaving
  - Continuations



Interfacing Varda with external world

Heterogeneous technologies
Varda agnostic




```
component Gateway {
    @expose
    string get(string key){
        (* Call the Router *)
    }
}
```

OTS interface

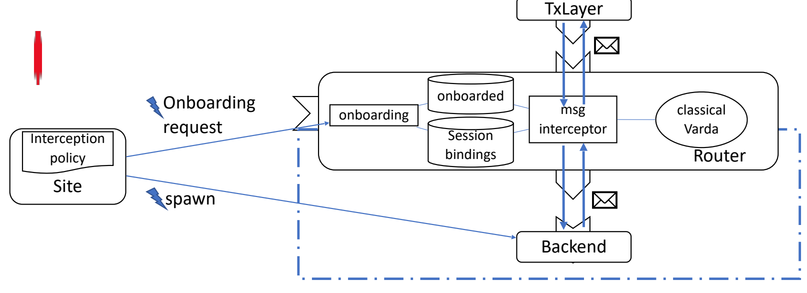
```
string get(string key);
contract get
returns value : string value-> value !=
"Error"

supervisionport redis_value =
        e ->print("Received");
```

Adaptor (+ Docker compose)

```
impl method get java:{=
  tmp = this.getJedis().get({{%key%}});
   if(tmp == null){
     return  "Error";
   else return tmp;
=}
```

Interception workflow

1. Preexisting architecture
2. User-defined interception
3. Static generation
4. Dynamic setup
5. Communication interception



Interception vs Parametrization

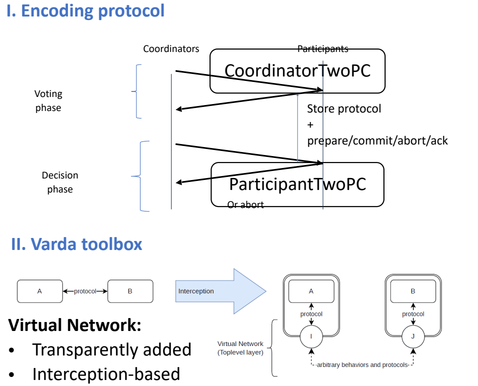
	Interposition	Parametrization
Evolution	Externally imposed (black box)	At the component level
Require		A priori design
Long term view	Ease hot swapping (e.g., K8S)	(metaprogramming in Varda?)

Evolution in Varda

- Architecture wide: Interception
- Local: Interface evolution

Encoding Two-Phase Commit

I. Encoding protocol

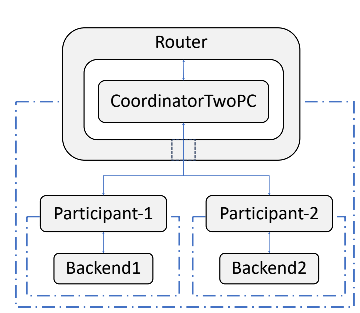


II. Varda toolbox

Virtual Network:

- Transparently added
- Interception-based

III. Transparent application



Supervision

How to react to failure ?

No out-of-the-box crash supervision

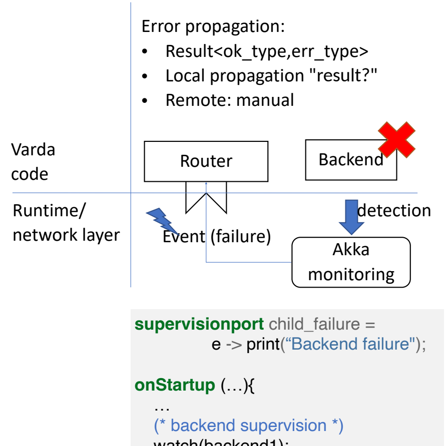
Explicit + Programmable

Reuse underlying runtime supervision

- Supervision ports
- Watch = runtime adaptor (target library)

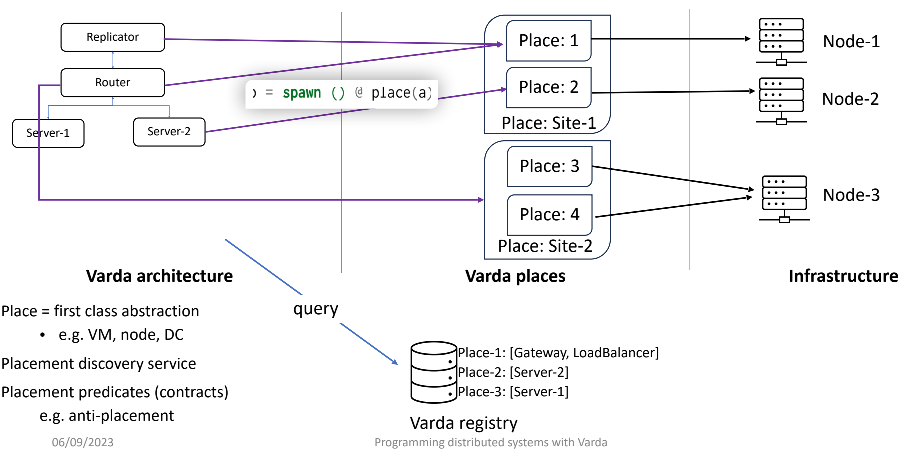
Error propagation:

- Result<ok_type,err_type>
- Local propagation "result?"
- Remote: manual




```
supervisionport child_failure =
        e -> print("Backend failure");

onStartup (…){
   …
   (* backend supervision *)
   watch(backend1);
}
```

Placement

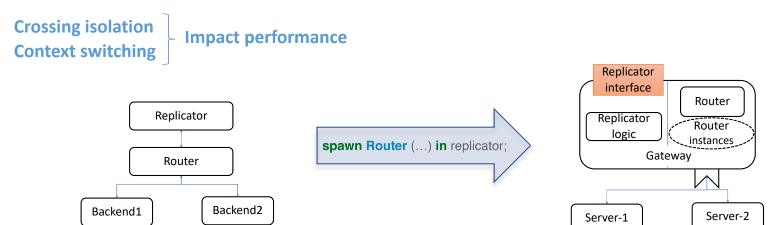


Place = first class abstraction
- e.g. VM, node, DC

Placement discovery service

Placement predicates (contracts)
e.g. anti-placement

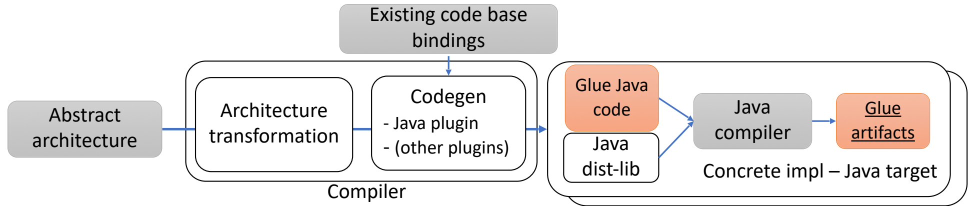
Inlining



Component inlining
- Loss of parallelism
- Preserves communication interface

How it works
- Static: inject Router code in Replicator
- Dynamic: instantiate router in replicator

Vardac: a compiler prototype



Architecture transformation
- Type checking
- Compile to simple Varda

Code generation
- Auditable => provenance
- Tracing + metrics

Compiler	OCaml (38 KLoC)
Java dist-lib	Java11 + Akka (4 KLoC)
Tests+benchmarks	Varda/Java (20KLoC)

Incremental building (VAntidoteDB)

	LoC architecture	LoC adaptors	Extension cost	OTS
Miscellaneous	219	29		
Single Shard, Single Site	710	184	No	Redis, Kafka, CRDT lib
Sharding	67	0	No	-
Strong consistent commit	159	0	No	-
Multi-Site	81	0	Logical clock type	RabbitMQ

! LoC with blank lines and comments