Programming Large-Scale Distributed Systems

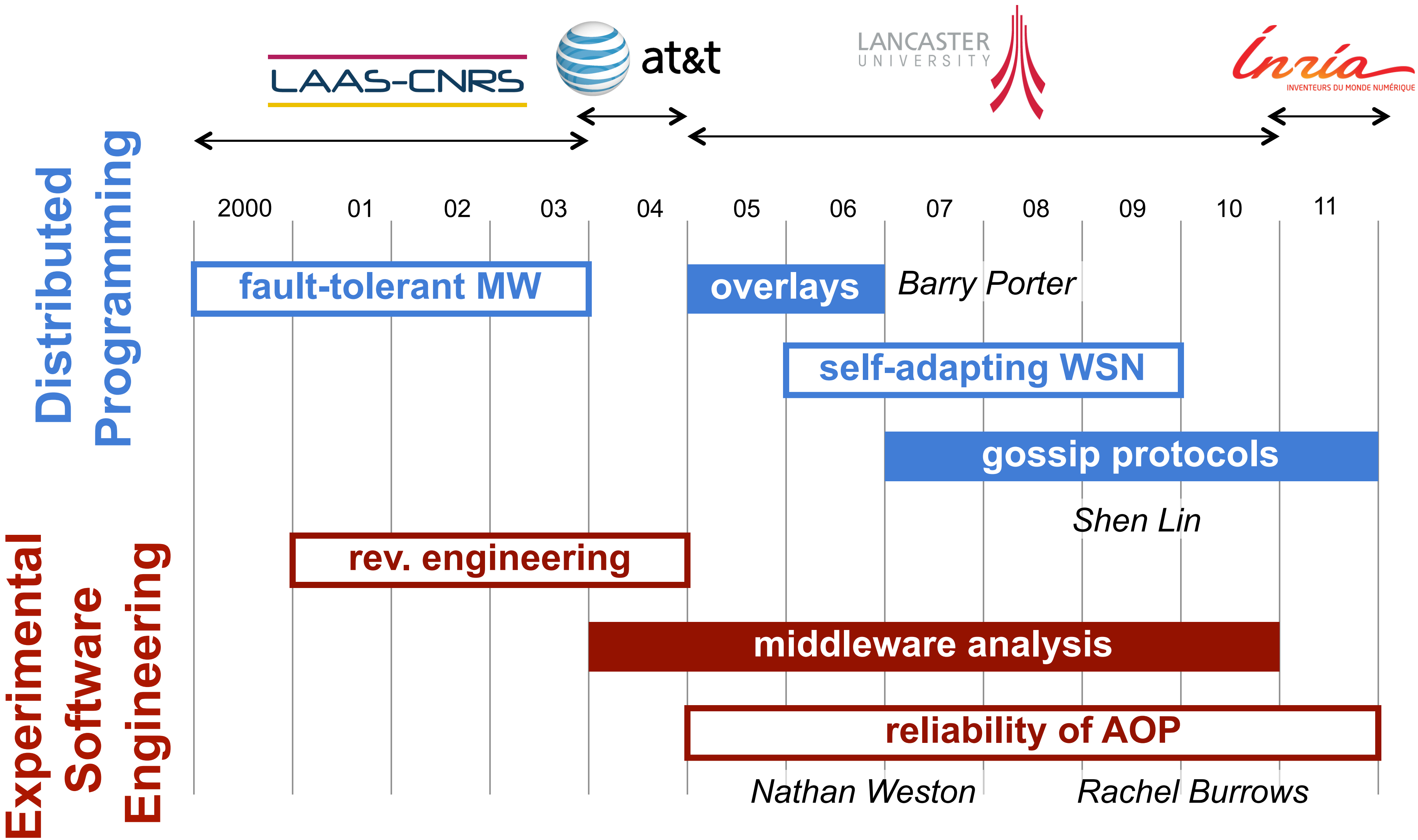
Some Mechanisms, Abstractions, and Tools

François Taïani

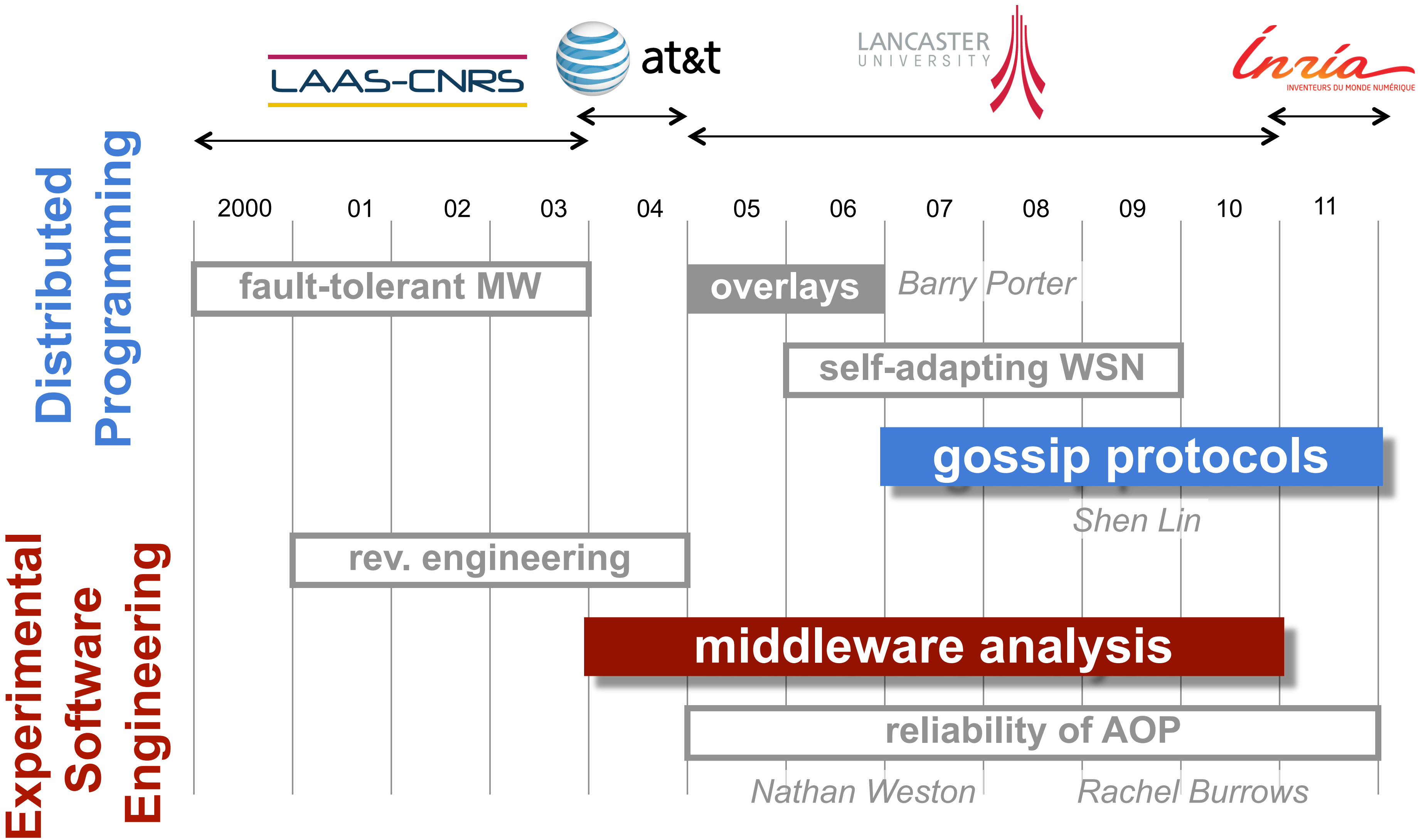
Soutenance d'HDR

17 Novembre 2011

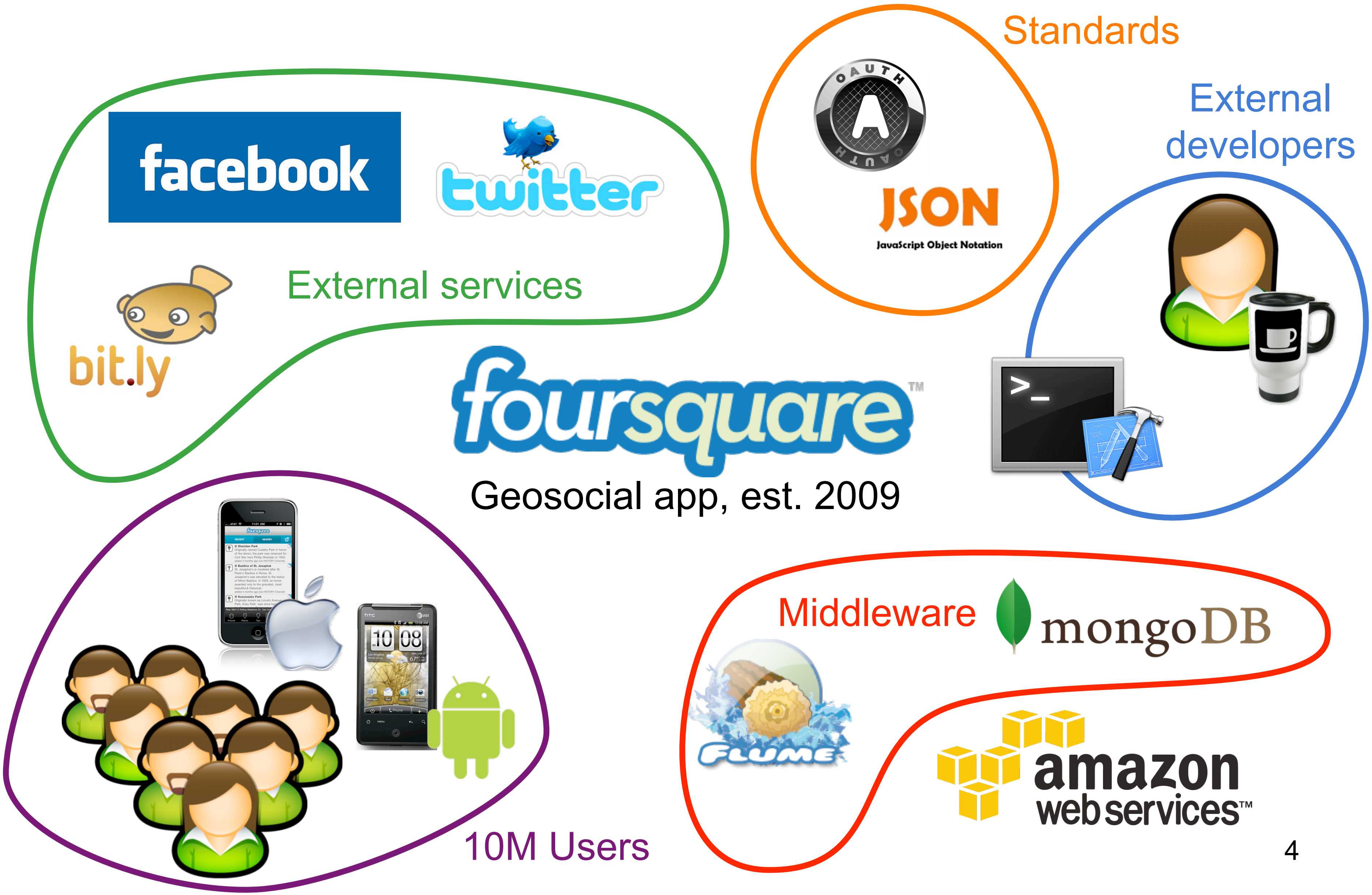
“Middleware Engineering”



“Middleware Engineering”



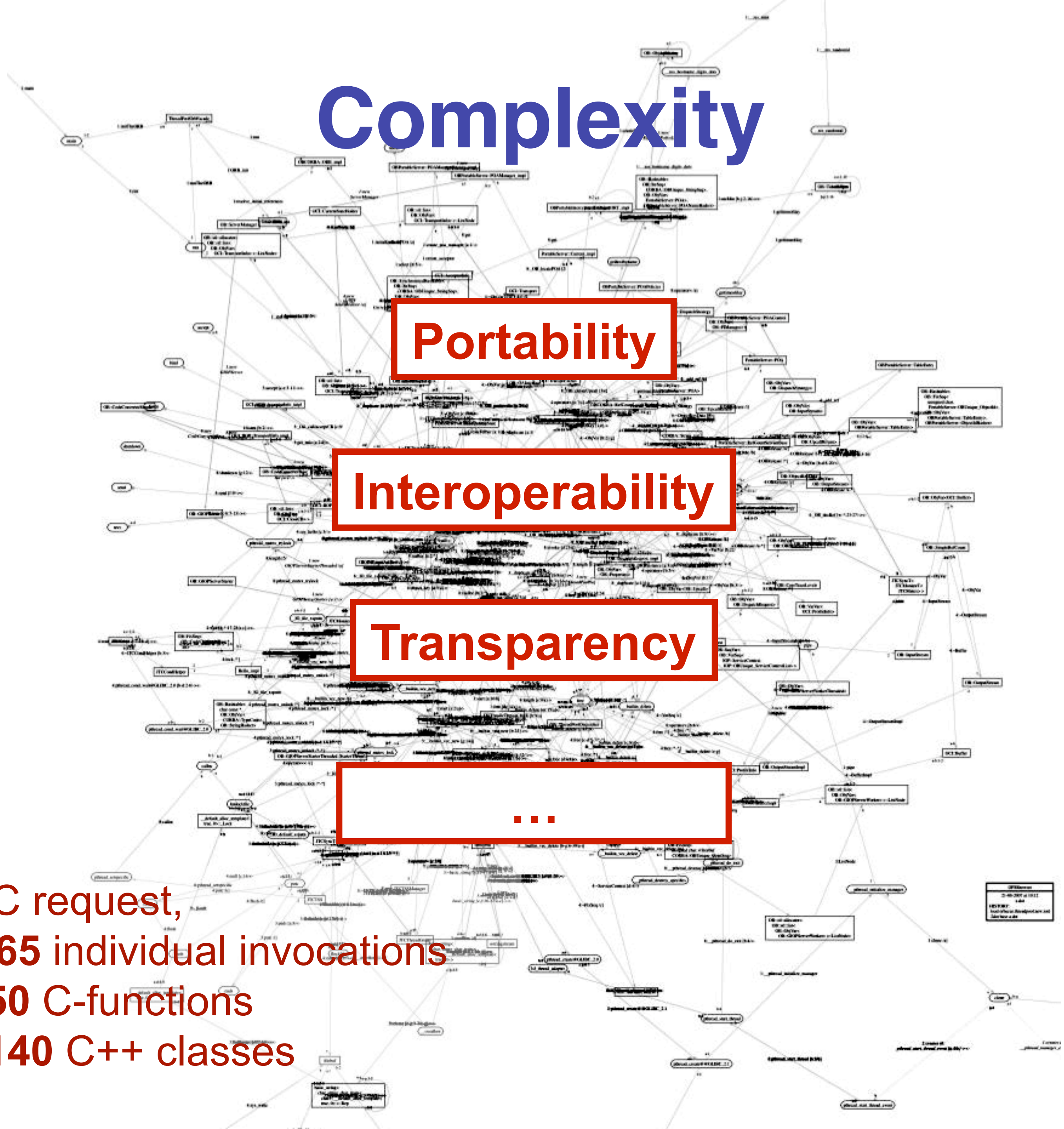
A Distributed System Today …



Challenges

- **Dynamicity & Scale**
  - ➔ Google ~ 1M (?) servers
  - ➔ foursquare (geosocial network): 10M users within 2 yrs
  - ➔ Facebook: 800M active users

Complexity



Portability

Interoperability

Transparency

…

one RPC request,

- **2065** individual invocations
- > **50** C-functions
- > **140** C++ classes

Challenges

- **Dynamicity & Scale**
  - ➔ Google ~ 1M (?) servers
  - ➔ foursquare (geosocial network): 10M users within 2 yrs
  - ➔ Facebook: 800M active users

- **Complexity & Heterogeneity**
  - ➔ ↗ functionalities
  - ➔ ↗ dependencies
  - ➔ ↗ providers
  - ➔ ↗ devices
  - ➔ ↗ inconsistencies
  - ➔ ↗ code size

How to **design**, **program**, and **analyse** these types of systems?

Our take

Reusable programming **abstractions**
for large-scale distributed systems

- Which **abstractions**?

- Supported by which **tools**?

Outline

- **Intro** (just done)
- **WhisperKit:**
  Programming Gossip-based Systems

- **ProfVis:**
  Anomaly Diagnosis in Grid Middleware

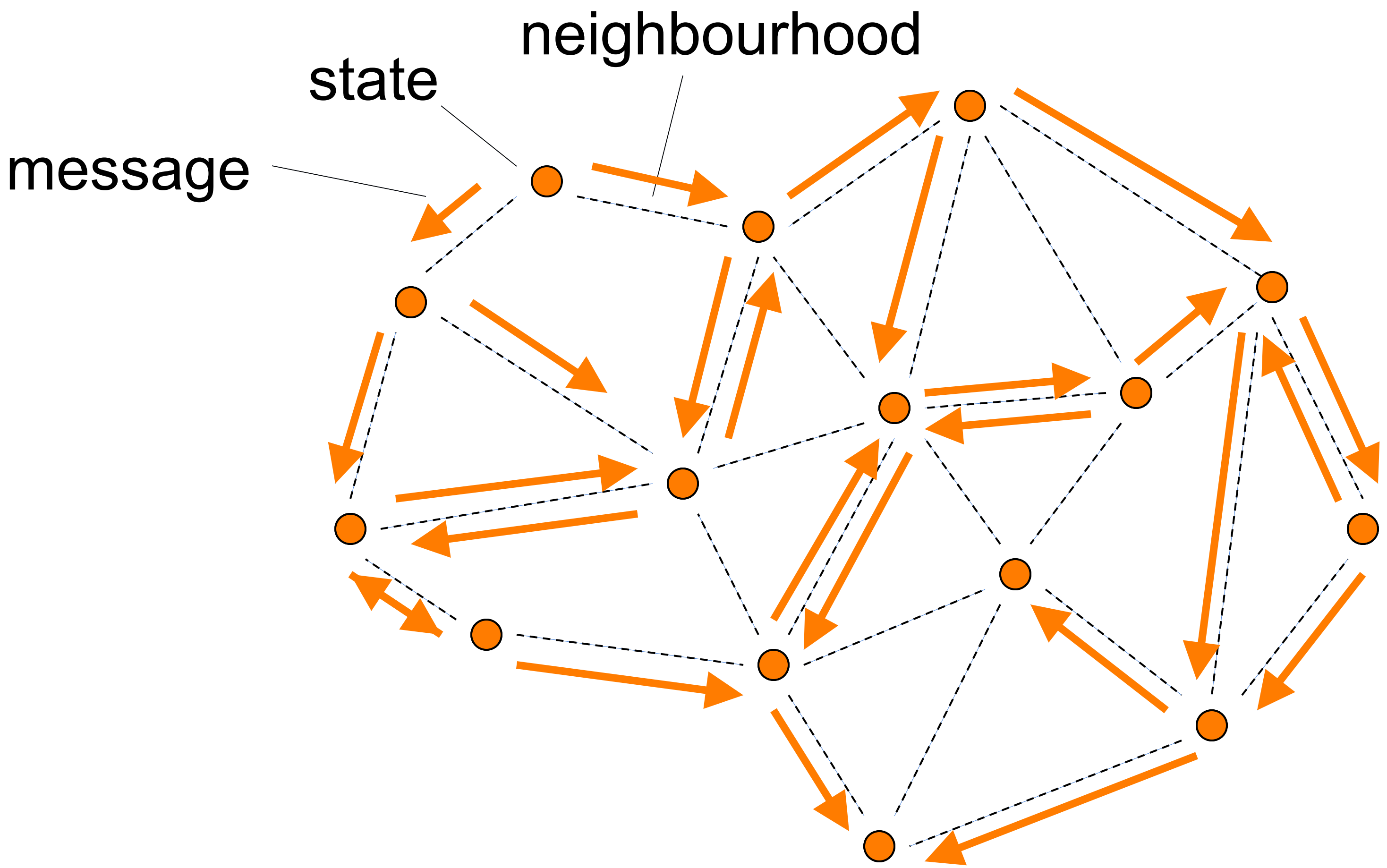
- **Conclusion and Outlook**

Outline

- **Intro** (just done)
- **WhisperKit:**
  Programming Gossip-based Systems 


- **ProfVis:**
  Anomaly Diagnosis in Grid Middleware 
- **Conclusion and Outlook**

Motivation: Gossip Protocols



- Highly **scalable**, **efficient**, and **robust**
  - ➔ Applied to wide range of services

The Problem with Gossip

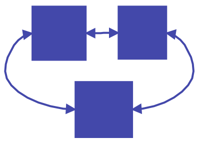
- Conceptually **simple**
  - ➔ typically symmetric behaviour
  - ➔ key notions of **state**, **decisions &** information **flows**

- But implementation can be **time consuming**

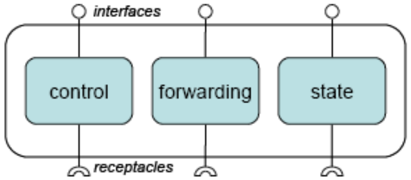
Which **reusable abstractions** to facilitate Gossip programming ?

Our Take: Components

- Component successfully applied to distributed systems
  - ➔ Rapidware, GridKit, Cactus, FraSCAti



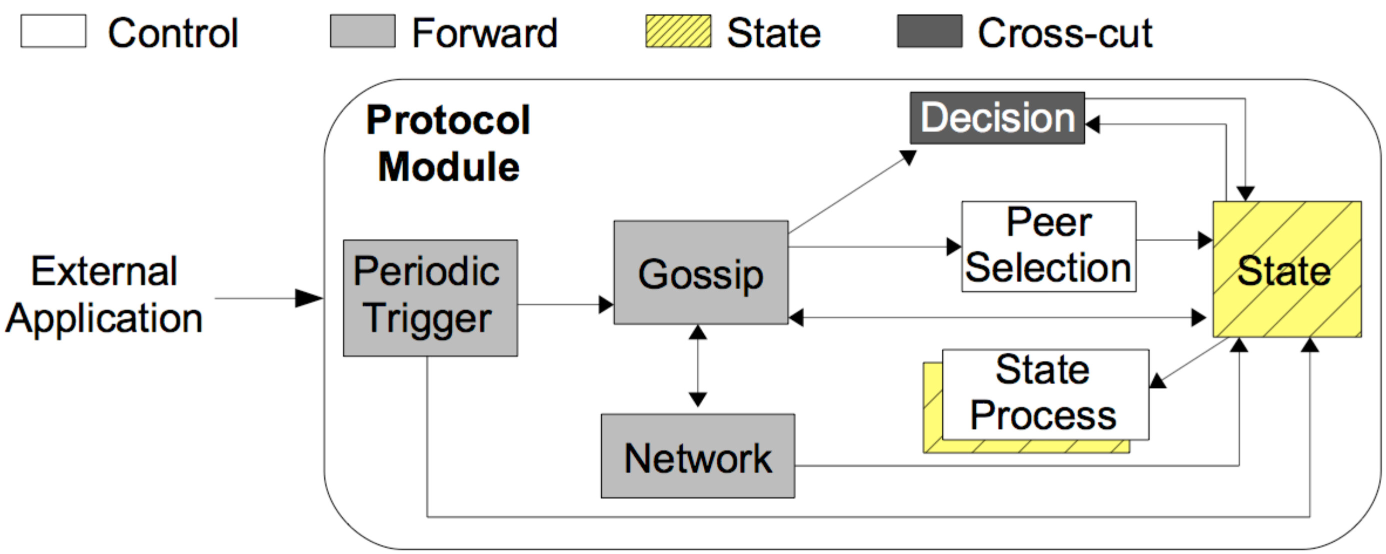
- Clear **structure**, explicit **dependencies**



- Benefits
  - ☺ **reusability**
  - ☺ **composability** and **configurability**
  - ☺ **runtime adaptation**

GossipKit

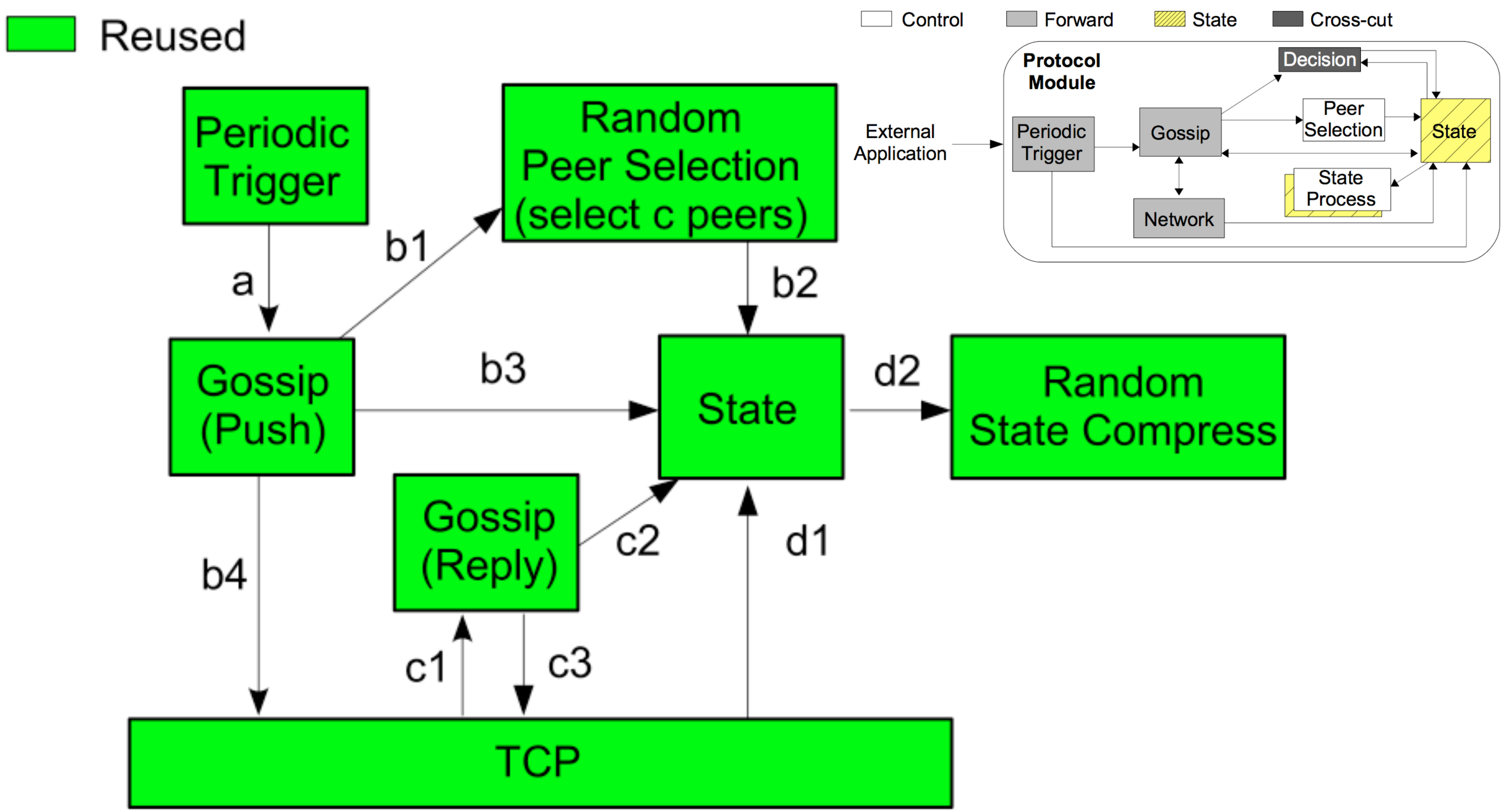
- Analysis of **30 Gossip protocols**



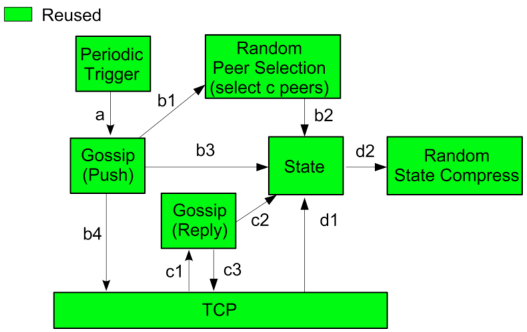
- Result: A component **framework** for **gossip** protocols
  - ➔ targets abstraction, reuse

Example: Random Peer Sampling

- **Goal**: periodically returns a random set of other peers

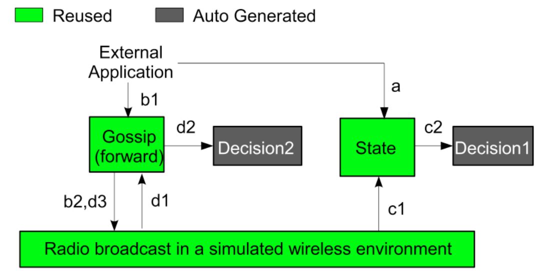


GossipKit Examples



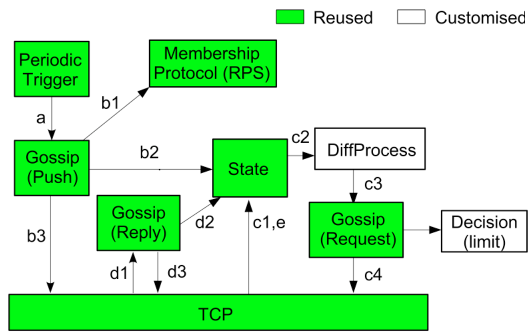
RPS

[ToCS 07]



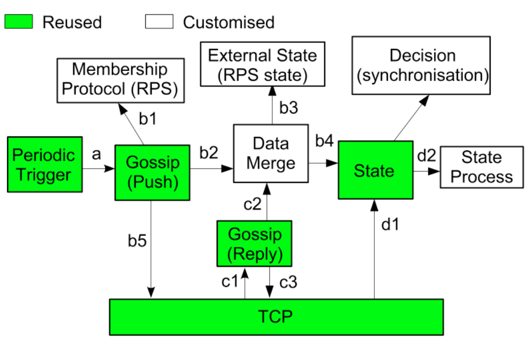
Wireless broadcast

[ToN 06]



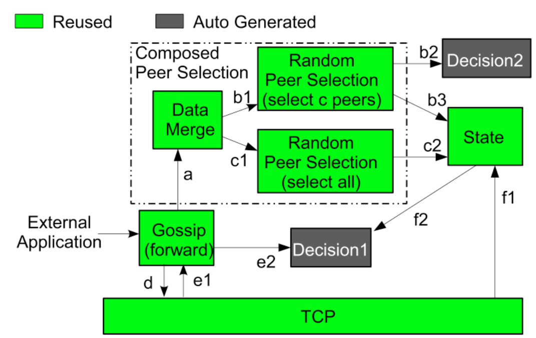
Anti-Entropy

[PODC 87]



T-Man

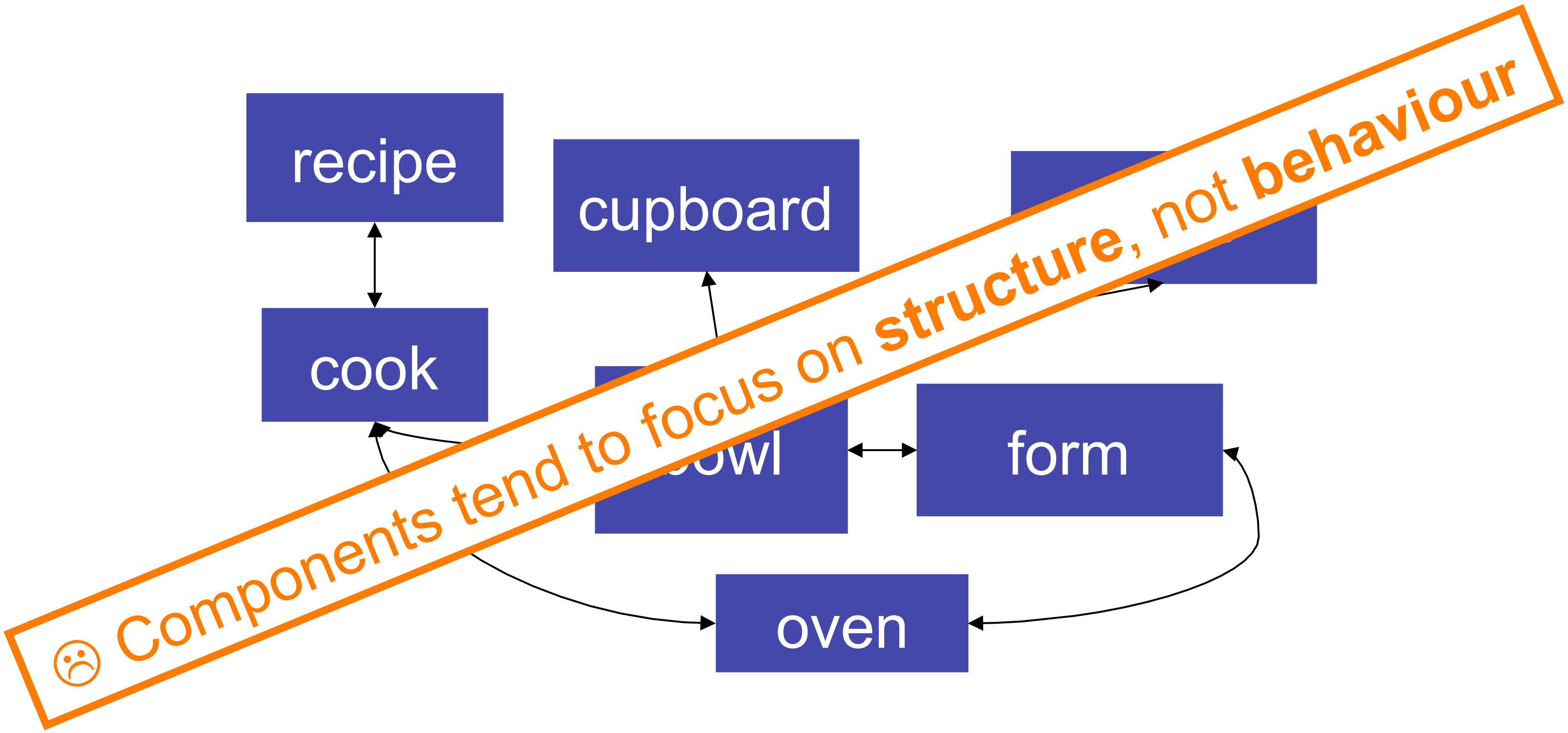
[Computer Networks 09]



SCAMP

[ToC 03]

The problem with components



How can best to combine **behaviour** and **structure**?

High-level dist. languages

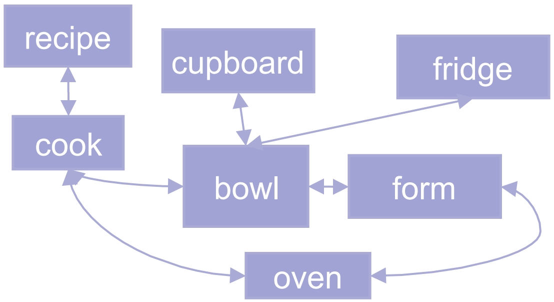
- **Spec. lang. and DSL:** High-level per node description
  - ➔ Lotos, Estelle, PLAN-P, Mace …

bake

- **Macro-programming**: system as one entity
  - ➔ E.g. Kairos, Regiment, TinyDB, MIT-Proto

- Benefits
  - ☺ high level of **abstraction** (in particular for macro-prog)
  - ☺ **intelligible**
  - ☺ good conceptual **match** for developers looking at behaviour

Behaviour rather than structure

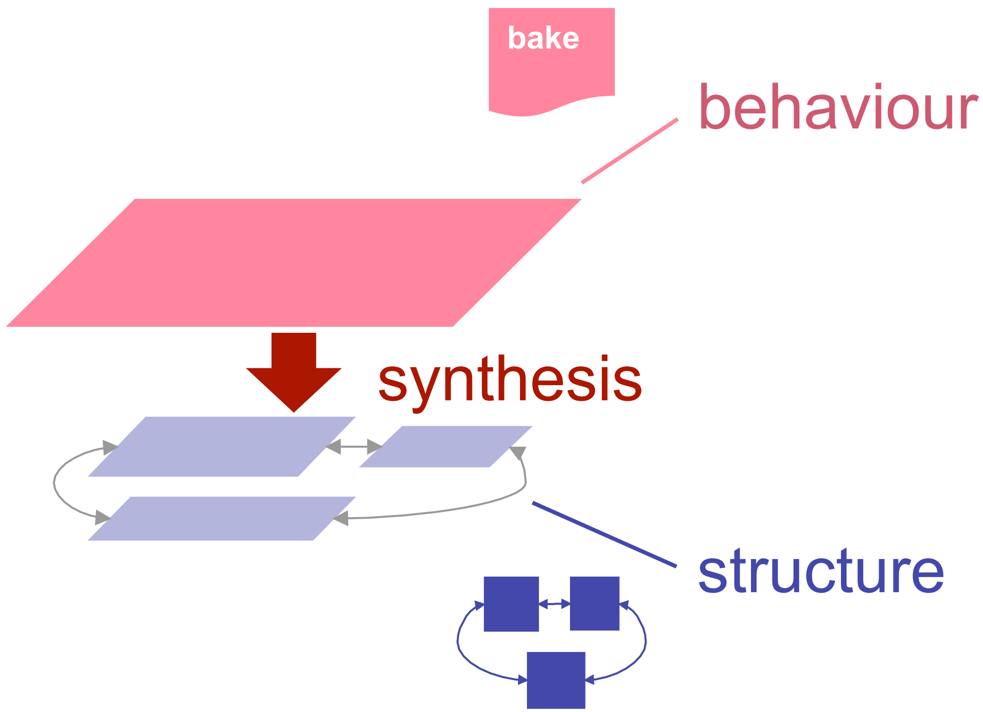



```
add(yohourt,1)
add(milk,2)
add(flour,3)
add(butter,1)
add(eggs,2)
add(soda)
bowl.mix()
bowl.pour(form)
form.putIn(oven)
bake()
```

- Drawbacks
  - ☹ we loose the benefits of components (reuse, adaptation, …)

Can we build a **hybrid approach** that combines the strengths of **components** & **high-level languages**?

Transparent Componentisation

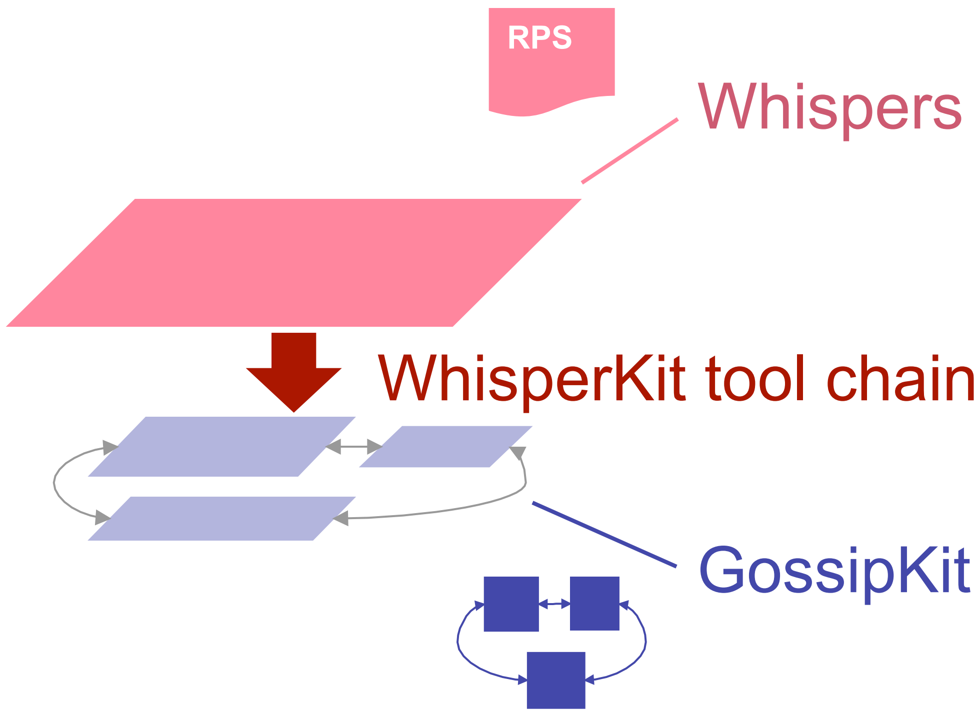


☺ simple
☺ concise
☺ high-level

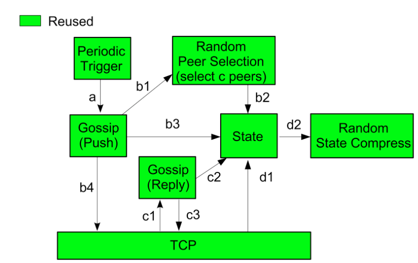
☺ modular
☺ reusable
☺ (re)configurable

- **Separation of concern** between behaviour / structure
- **Developers** can focus on **high level logic**
- **Systems** takes care of **modularity**, reuse, and evolution

WhisperKit = Whisper + GossipKit




```
RPS {
  State sample = …
  Node n, i;
  every (5000) {…}
}
```

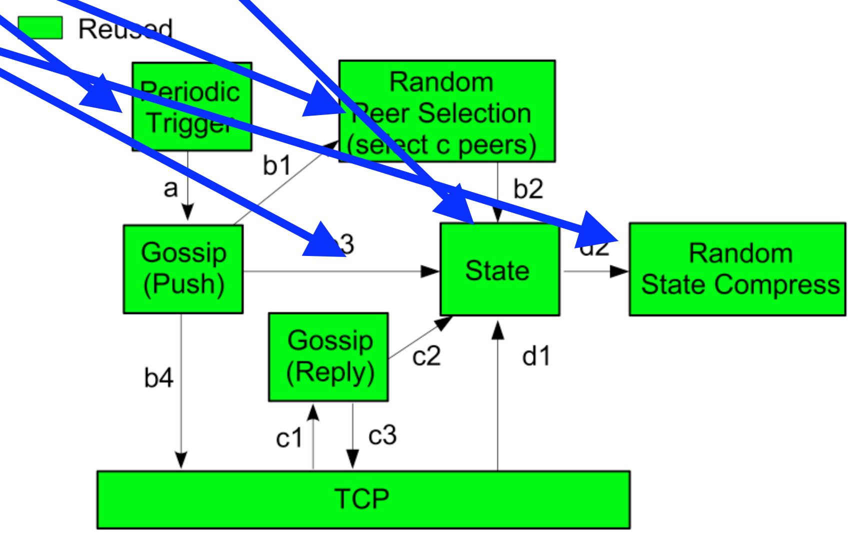


- **Whispers:** inspired from macro-programming (Kairos,…)
- **WhisperKit:** compiler/deployment chain (JavaCC)
  - ➔ Built-in support for distributed reconfiguration

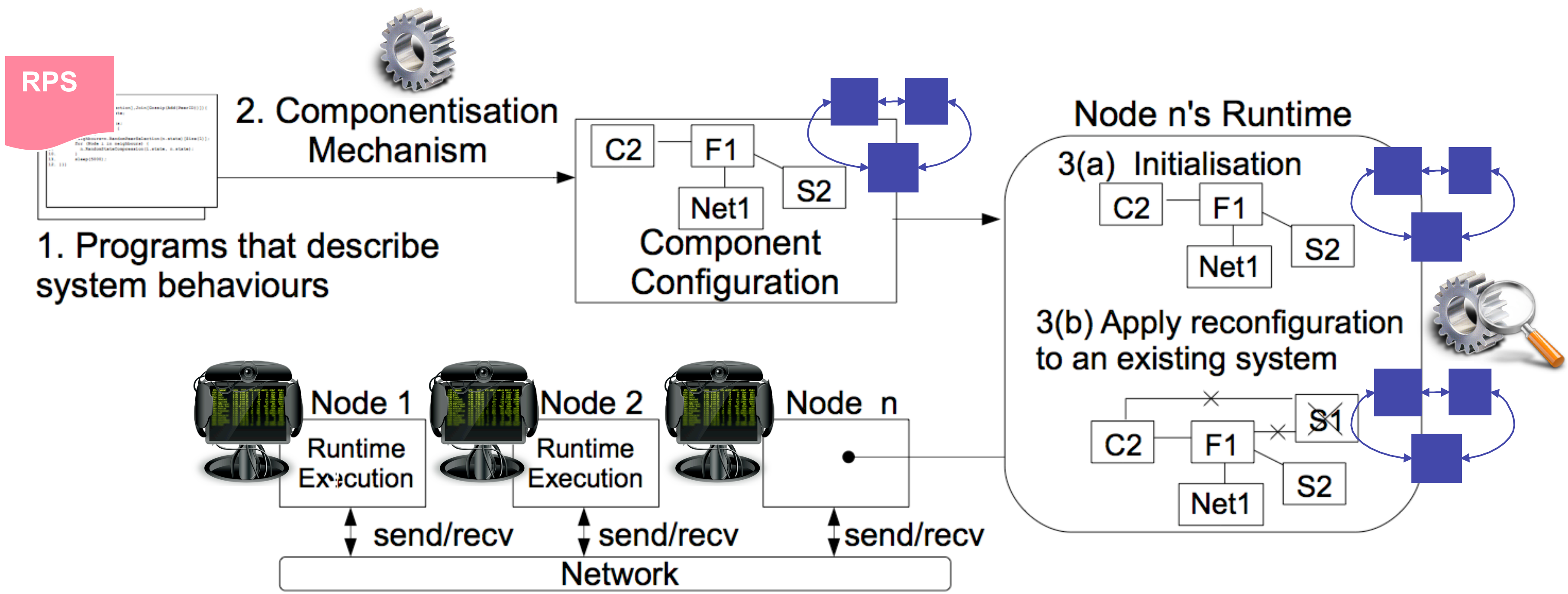
Whispers Example: RPS

RPS

```
RPS {
   State sample = new State[Node:PeerID][Size=5];
   Node n, i;
   every (5000) { // do the following every 5000 ms
      foreach (n in AllNodes) { // for each node n
         i=n.RandomPeerSelection(n.sample)[Size=1];
         n.sample.add([n]);
         i.RandomStateCompress(i.sample,n.sample)[Size=5];
         n.RandomStateCompress(i.sample,n.sample)[Size=5];
      } // end of foreach
   } // end of every
} // end of RPS protocol
```

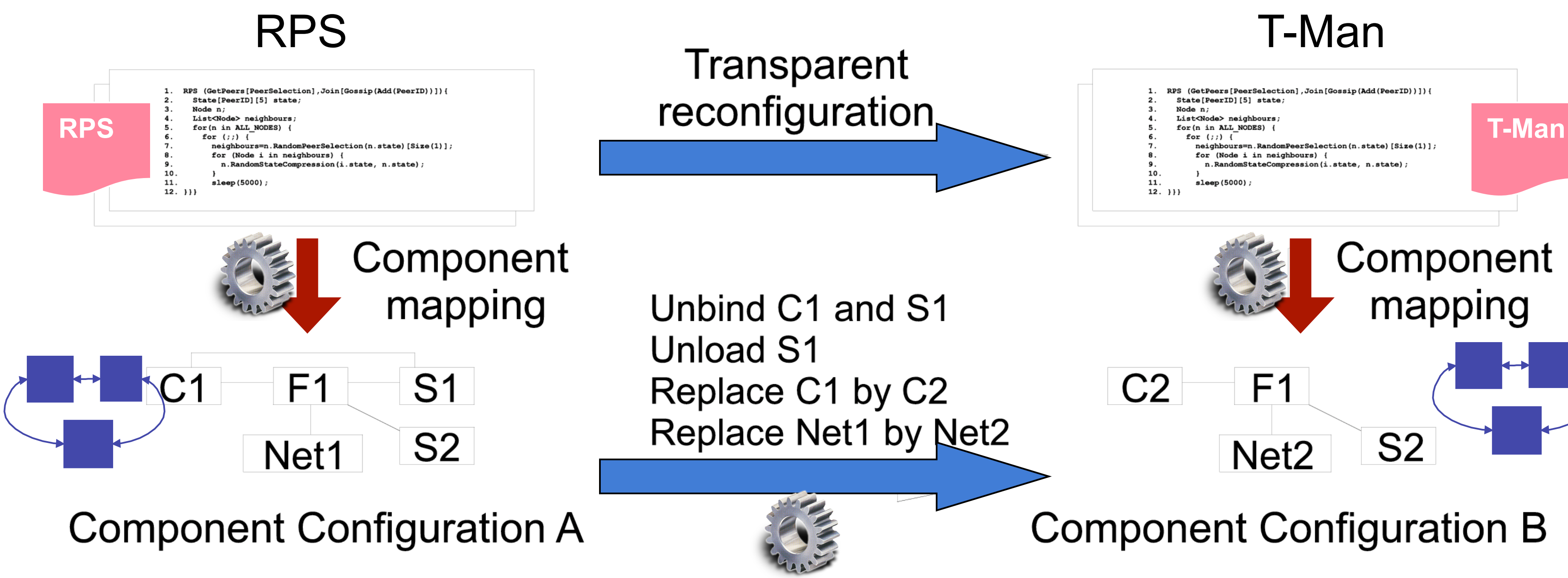


Deployment Process



Distributed Reconfiguration

- Developers describes new behaviour in Whispers
- Platform uses component representation
  - → to compute minimal set of changes
  - → to propagate and enact reconfiguration



Distributed Reconfiguration

- Example: RPS → T-Man(Ring) → T-Man(Grid)

coarse grained *fine grained*

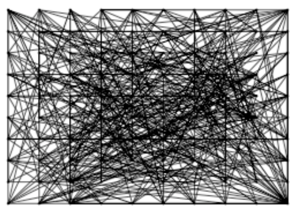


Figure 5.6: Initial random graph maintained by RPS

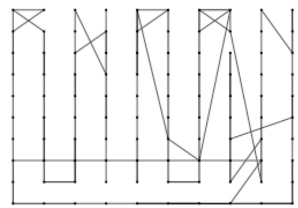


Figure 5.7: 5th rounds since 1st reconfiguration

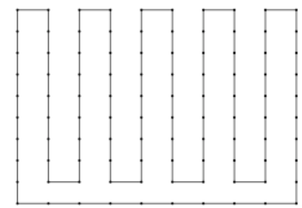


Figure 5.8: Ring constructed at the 11th round

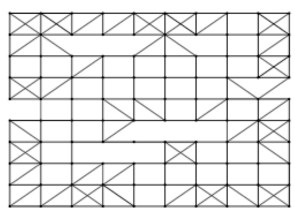


Figure 5.9: Topology at the 20th round

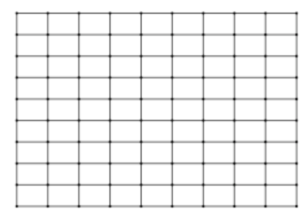


Figure 5.10: Grid constructed at the 23rd round

Evaluation: Simplicity (1)

Protocol	WHISPERS	Java	GOSSIPKIT XML Configuration
`Gossip1`	14	277	39
`Gossip2`	14	279	39
Anti Entropy	16	544	100
Averaging	14	466	85
Ordered Slicing	14	471	85
RPS	12	439	81
SCAMP	19	463	88
T-Man	20	491	93
Average	15.4	424	76.3

Evaluation: Simplicity (2)

Protocol	WHISPERS	**Java**	GOSSIPKIT **configuration**		
	Cyclomatic Comp.		Component	Parameter	Connection
Gossip1	2	11	5	6	7
Gossip2	2	11	5	6	7
Anti Entropy	3	10	9	15	13
Averaging	3	6	8	12	11
Ordered Slicing	3	11	8	12	11
RPS	2	12	7	15	10
SCAMP	3	20	8	10	12
T-Man	3	11	8	15	12
Average	2.6	11.5	7.3	11.4	10.4

Cyclomatic Complexity [McCabe76]:
≈ Number of decision points within a program

Summary

- **GossipKit**
  - ➔ First component-based framework for gossip protocols
  - ➔ Simple and general
- **Whispers/WhisperKit** (CBSE + DSL)
  - ➔ separates behavioural from structural concerns
  - ➔ Highly concise programs, that retain component benefits
- **Impact** of this line of research
  - ➔ one thesis
  - ➔ collaboration links with INRIA Rennes
  - ➔ publications at ACM SAC, DAIS'08, DAIS'09
  - ➔ Available on line: http://ftaiani.ouvaton.org/GossipKit/

Outline

- **Intro** (just done)
- **WhisperKit:**
  Programming Gossip-based Systems

- **ProfVis:**
  Anomaly Diagnosis in Grid Middleware

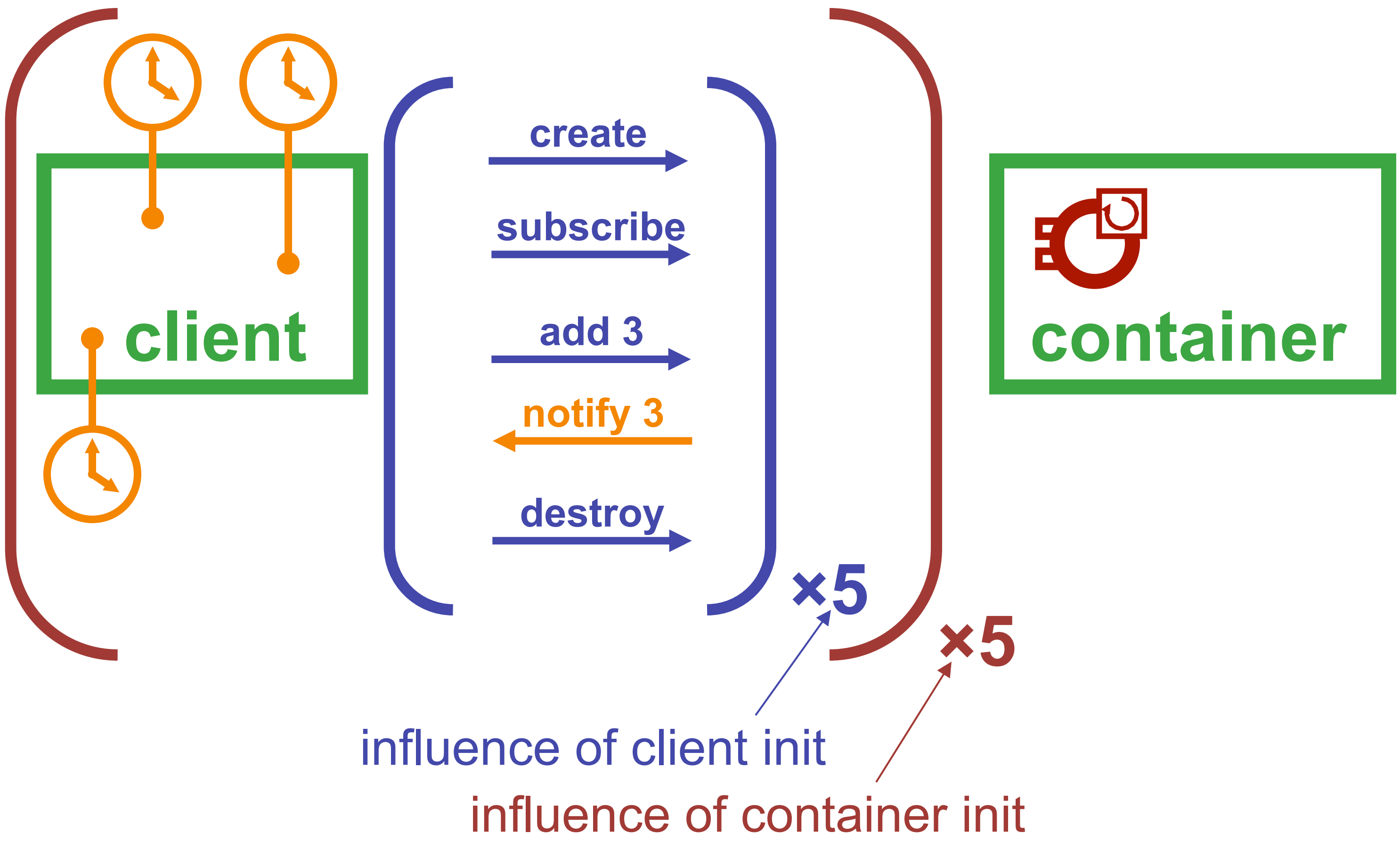
- **Conclusion and Outlook**

Studying Real-Life Reuse

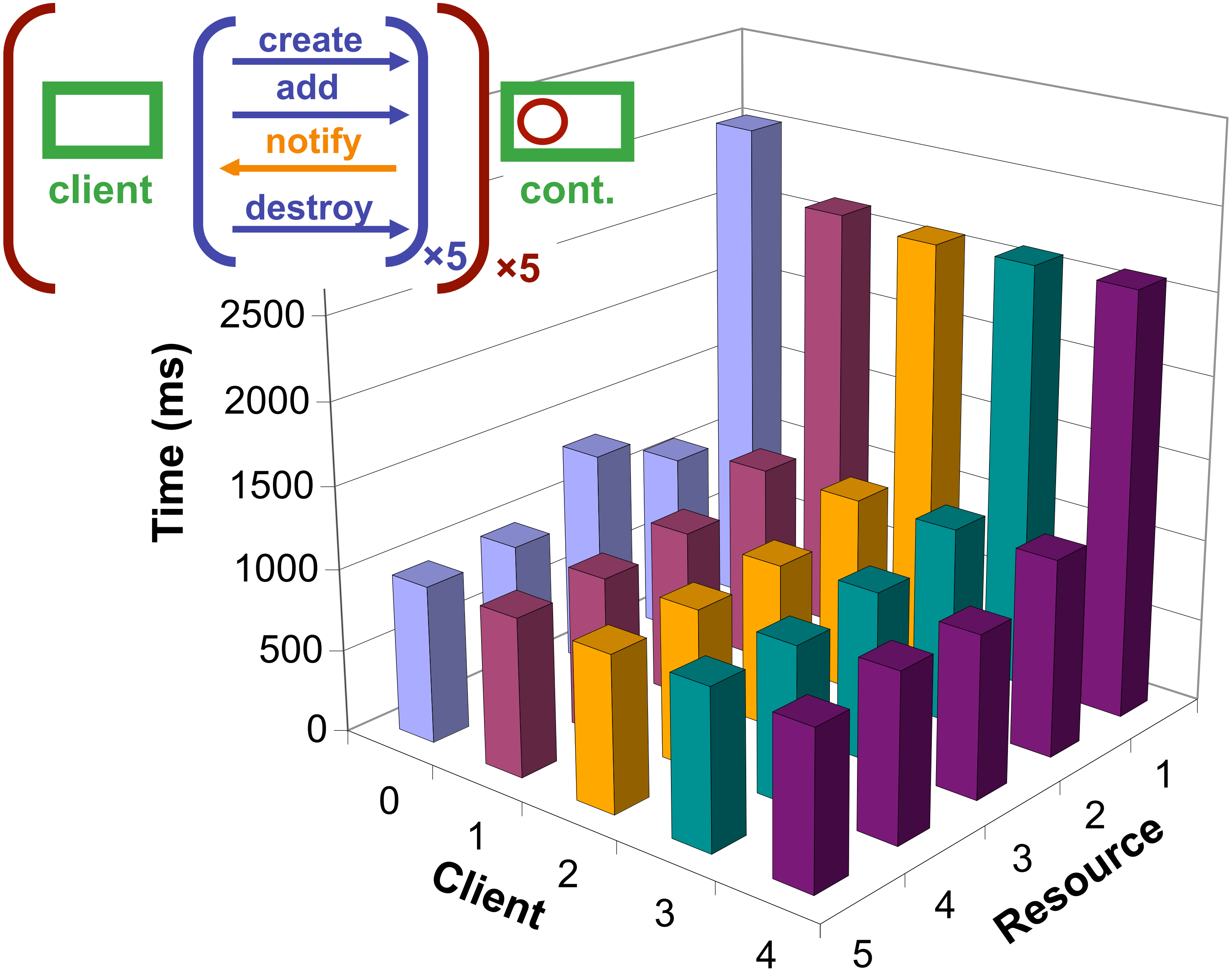
- **Globus** (Argonne): ref. implementation for Grid
  - → Grid Computing + Web Services
- Transition to WS stack (Version 3.9.x, 2005)
  - → within a short time (a few months)
  - → large, complex, collaborative (IBM, Apache,...) ↗ reuse
- **But … poor performances**
  - → Up to **30s** to **create** a simple distributed object (counter)
  - → Up to **2s** for a roundtrip remote **add** operation

- **Where** does these poor performances come from?
- **What** does it tell about modern MW development?

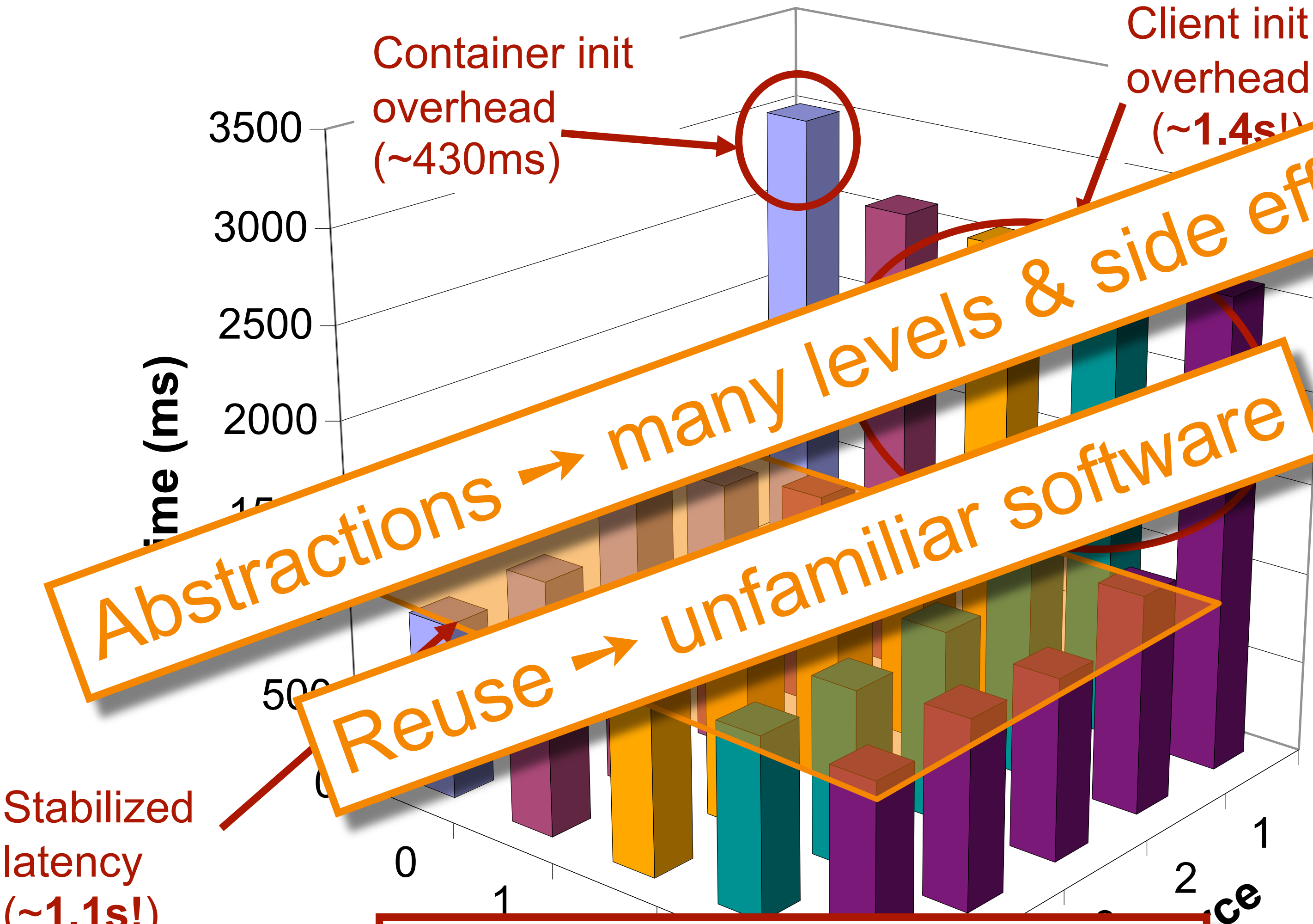
Experience 1: Initialisation



Finding 1: Init is a killer



Finding 1: Init is a killer

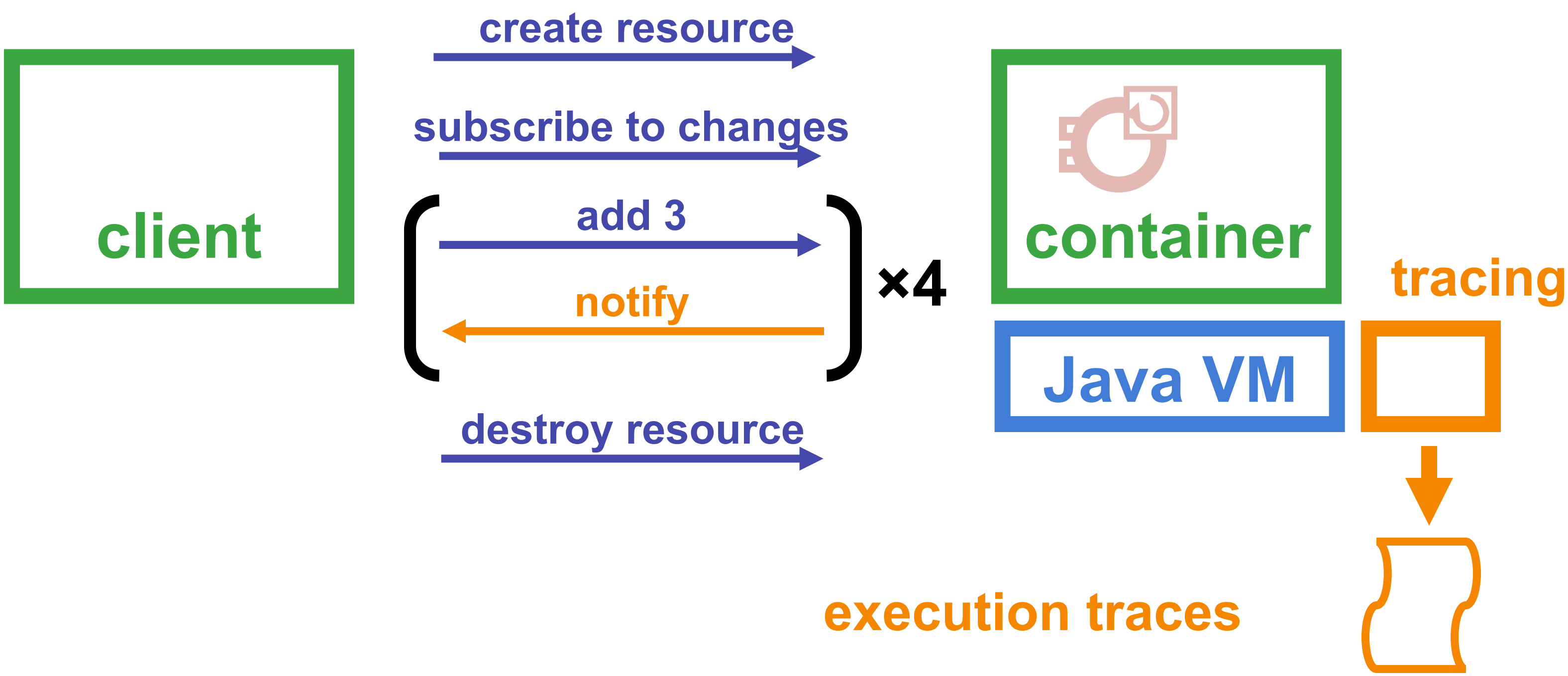


Abstractions → many levels & side effects

Reuse → unfamiliar software

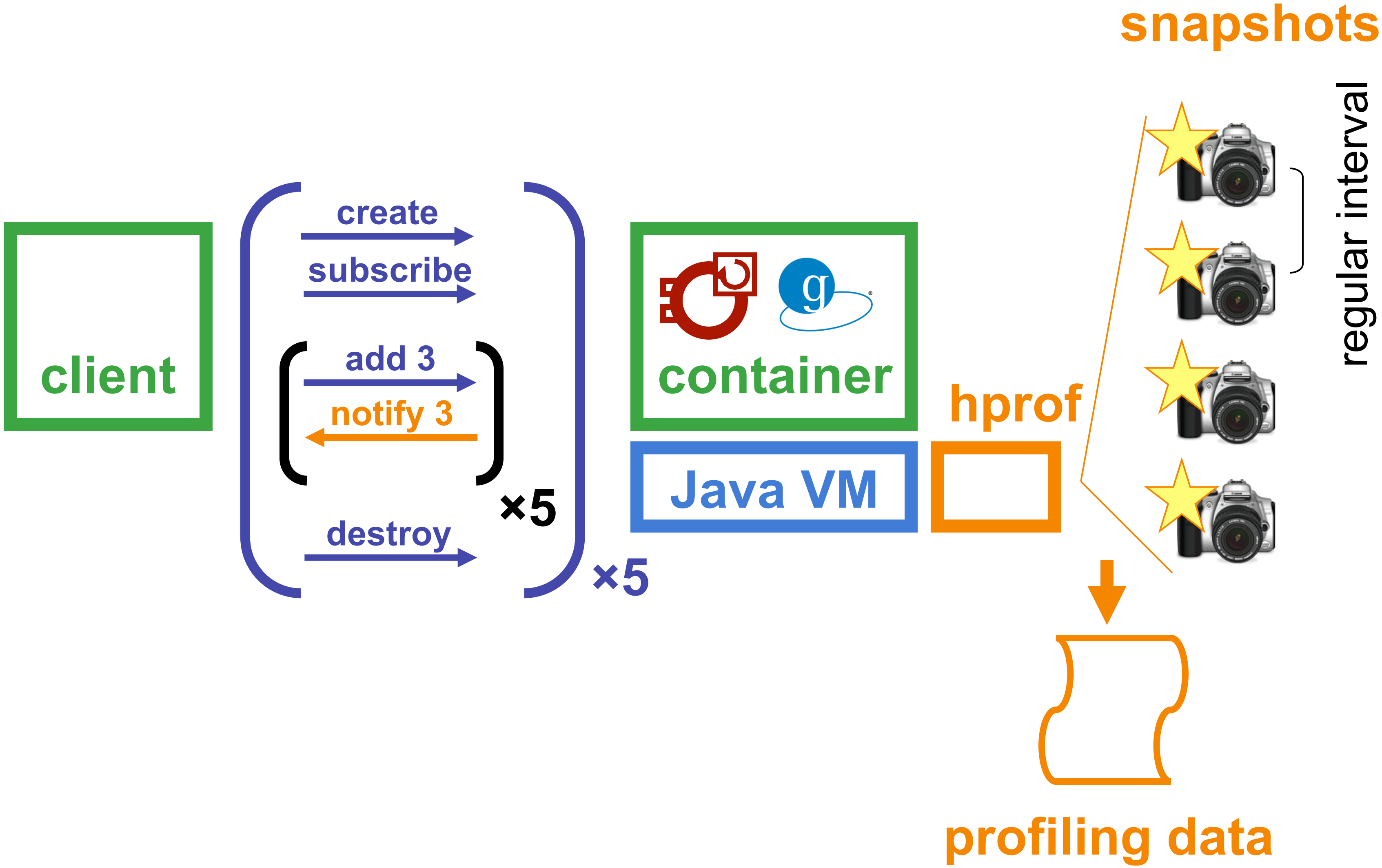
How to analyse this?

Exhaustive Tracing Intractable

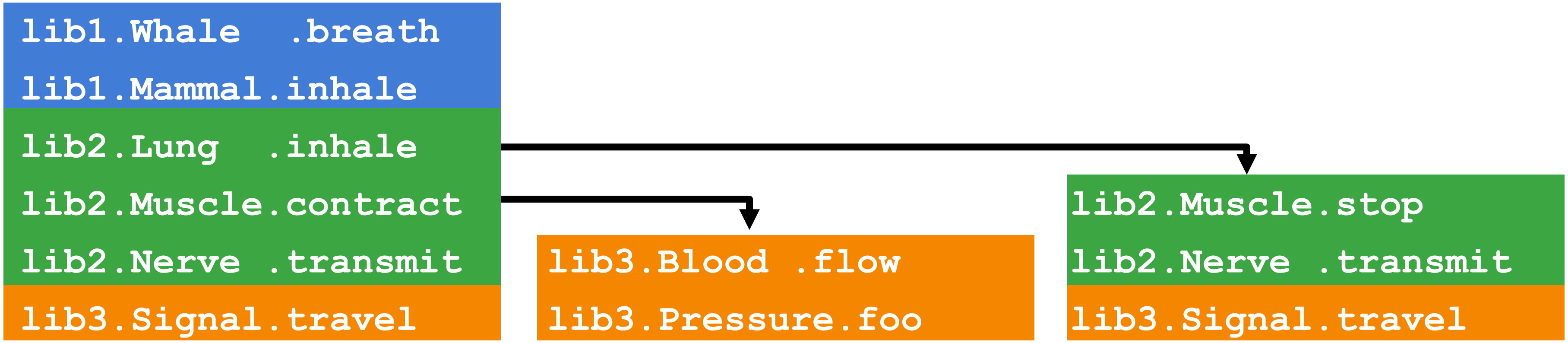


- First attempt: tracing everything (outside the JVM libs)
  - ➔ client : **1,544,734** local method call (sic)
  - ➔ server : **6,466,652** local method calls (sic) [+time out]
- **How to work around this data explosion?**

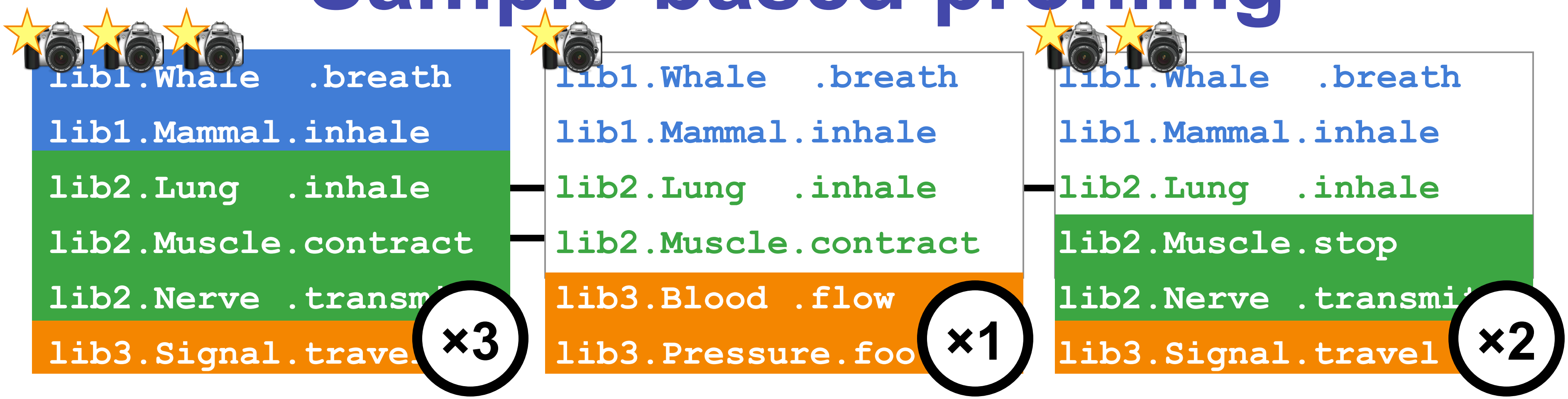
Sample-based profiling



Sample-based profiling



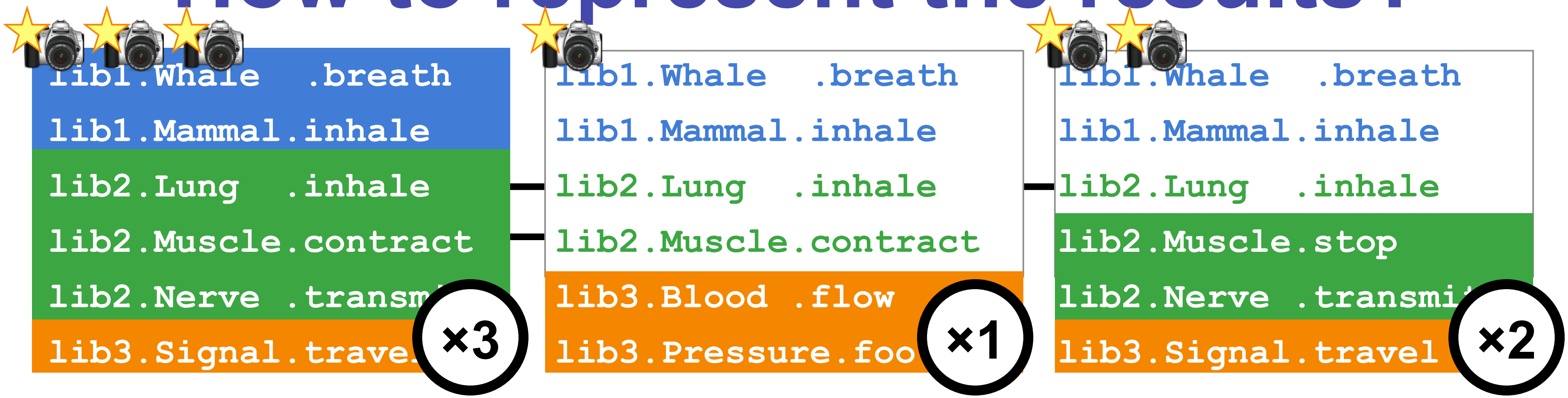
Sample-based profiling



Sampling yields a set of weighted stack traces (weight reflects time spent)

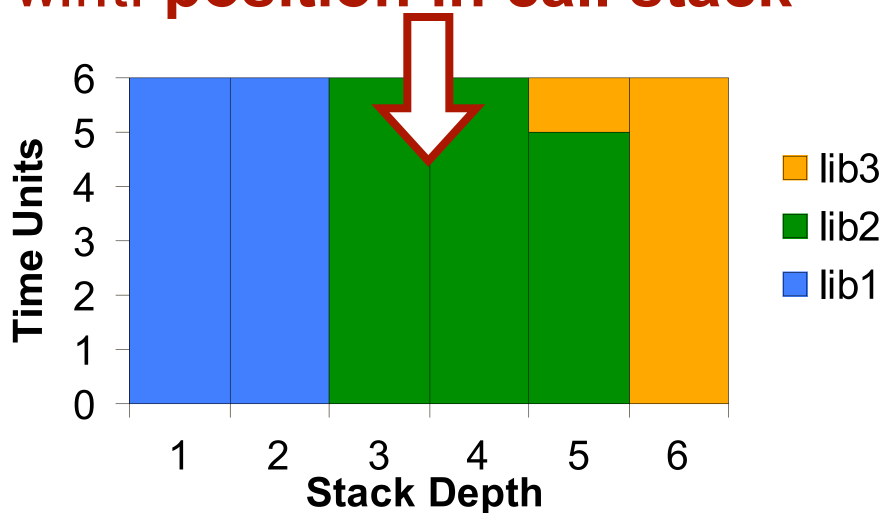
- Problem: **Data explosion.** On Globus :
  - ➔ 55550 method invocations
  - ➔ 1861 methods
  - ➔ 724 classes
  - ➔ 182 Java packages.
  - ➔ 32 threads

How to represent the results?

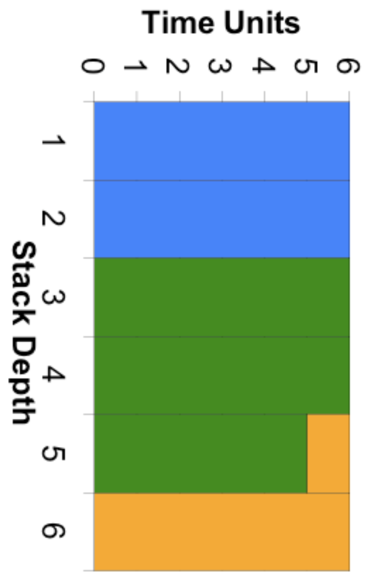


Sampling yields a set of weighted stack traces (weight reflects time spent)

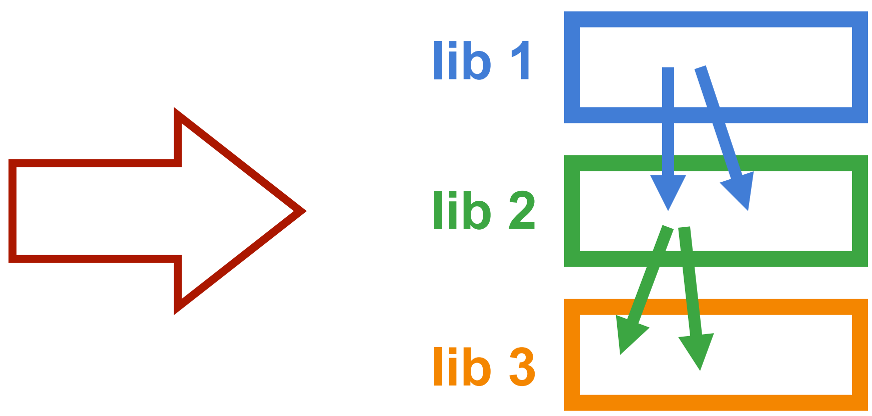
- **Aggregates** invocations of the same library
- Chart w.r.t. **position in call stack**



How to represent the results?

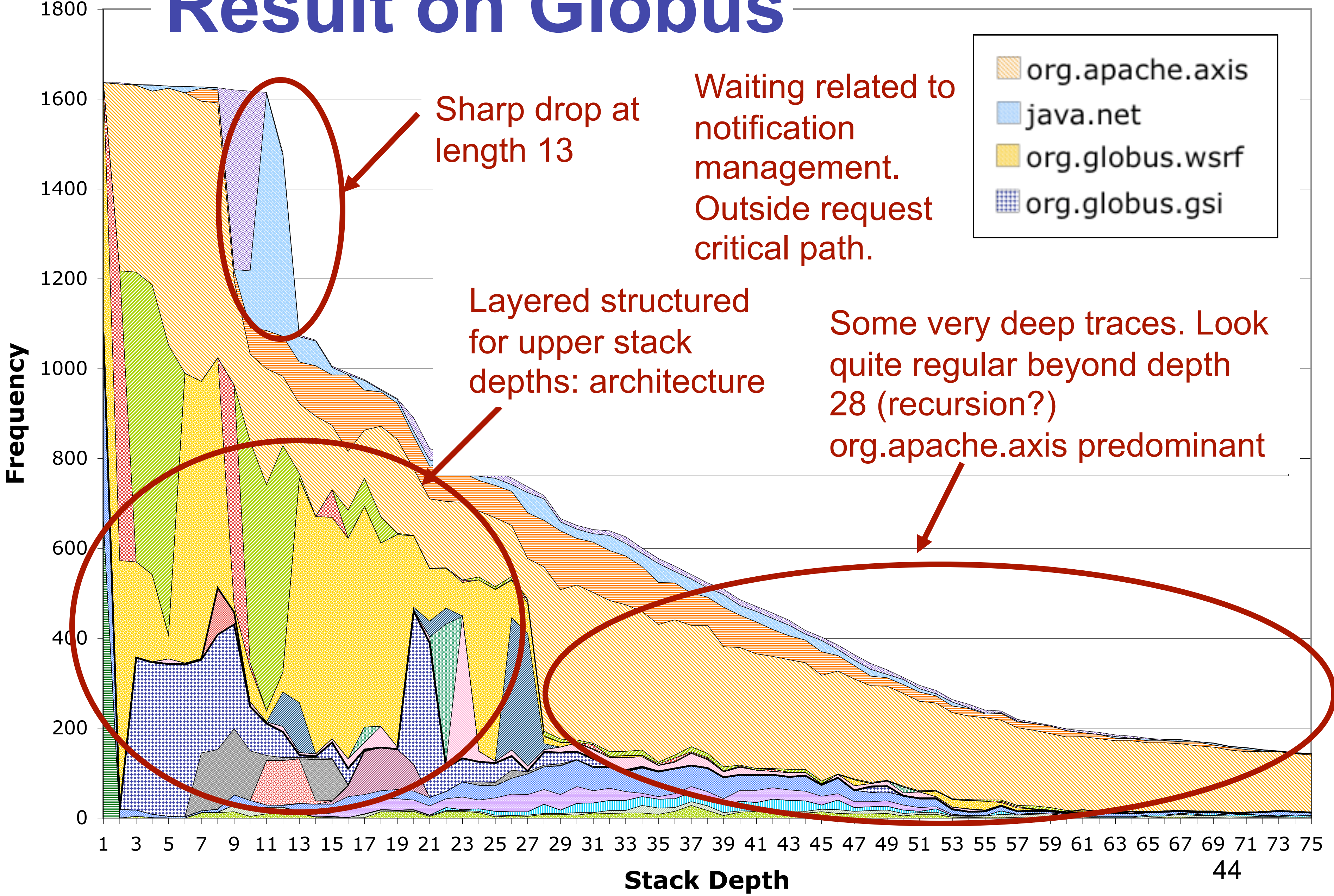


Package Activity vs. Stack Depth



Software Structure

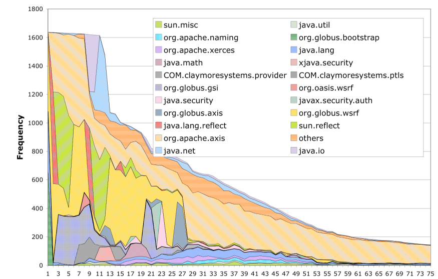
Result on Globus



Findings

- **XML management issue** in apache.axis.wsdl
  - ➔ very deep recursion involving one method
- No clear culprit for overall performance
  - ➔ **Axis** 37%
  - ➔ **SOAP + XML** 44%
  - ➔ **Security (GSI, RSA)** 30%
- More generally, typical example of
  - ➔ **deep** analysis
  - ➔ in **unfamiliar software**

Interactive Visualisation

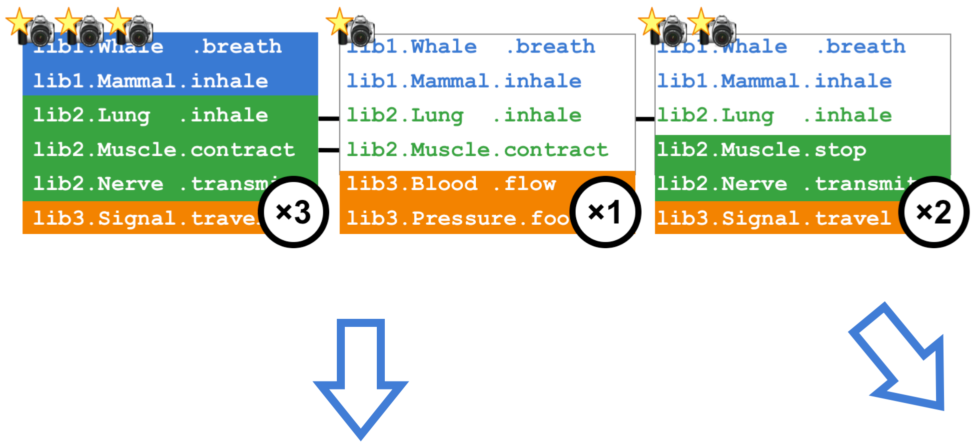


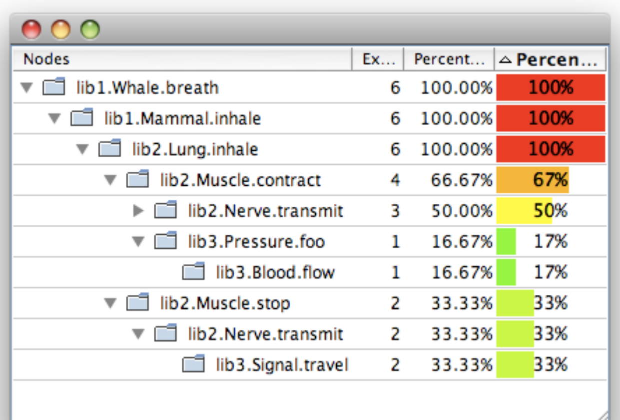
- Problem: stack depth project is **static**
  - ➔ call relationships hidden, compaction fixed
- Our take: **interactive navigation**
  - ➔ use **structural information** in dynamic data
    e.g. org.apache.axis.utils.ClassUtils.forName()
  - ➔ vary **'local abstraction'** level at which data is shown

?

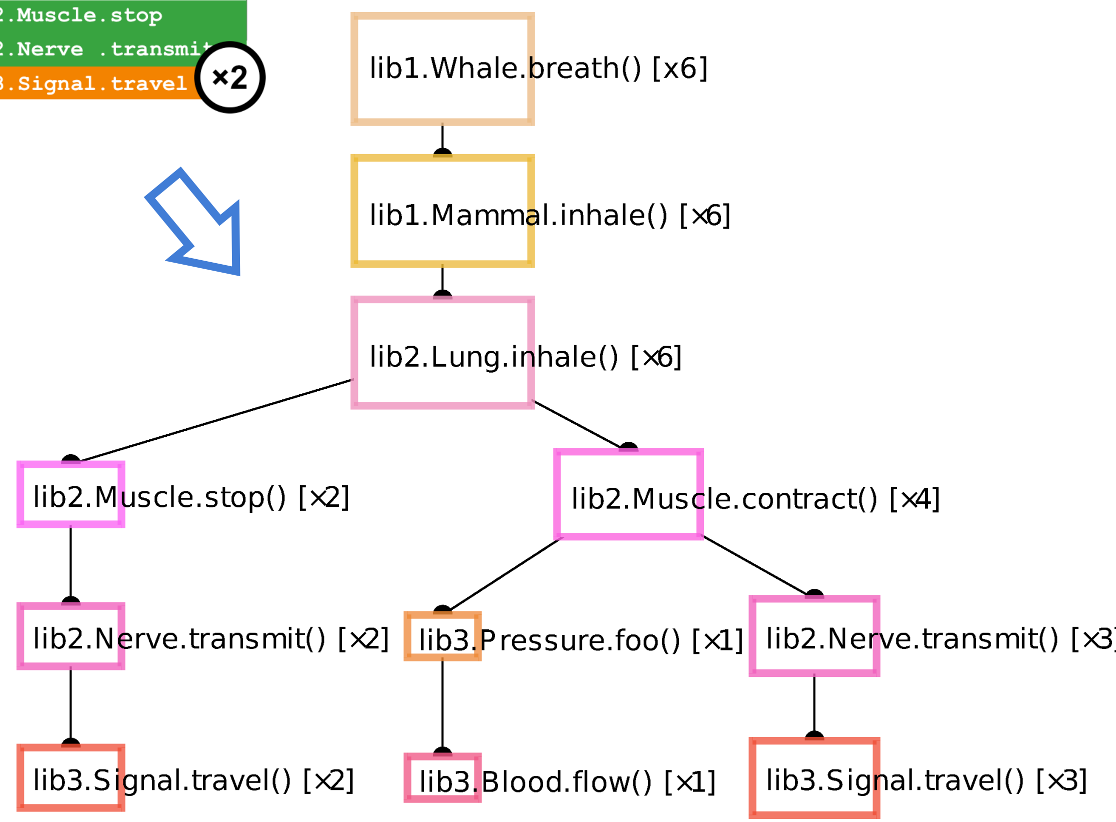
- Result: collaboration with Psychology Dpt (Lancaster)
  - ➔ application of structural compaction to **dynamic data**
  - ➔ **ProfVis prototype** and explorative user study

Back to biology example



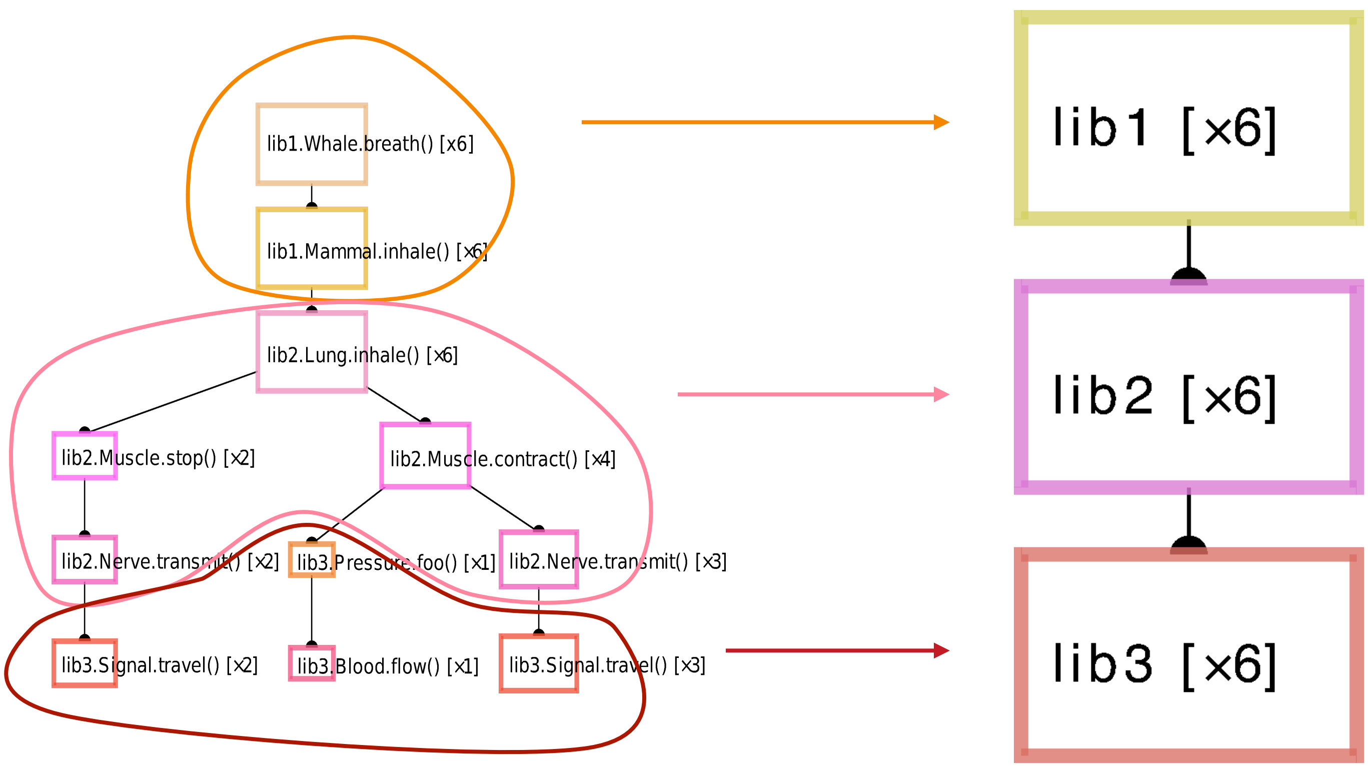


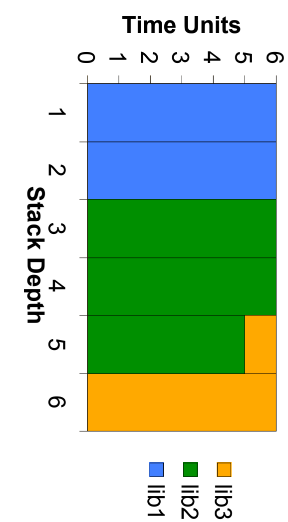
Nodes	Ex...	Percent...	Percen...
lib1.Whale.breath	6	100.00%	100%
lib1.Mammal.inhale	6	100.00%	100%
lib2.Lung.inhale	6	100.00%	100%
lib2.Muscle.contract	4	66.67%	67%
lib2.Nerve.transmit	3	50.00%	50%
lib3.Pressure.foo	1	16.67%	17%
lib3.Blood.flow	1	16.67%	17%
lib2.Muscle.stop	2	33.33%	33%
lib2.Nerve.transmit	2	33.33%	33%
lib3.Signal.travel	2	33.33%	33%



Full compaction

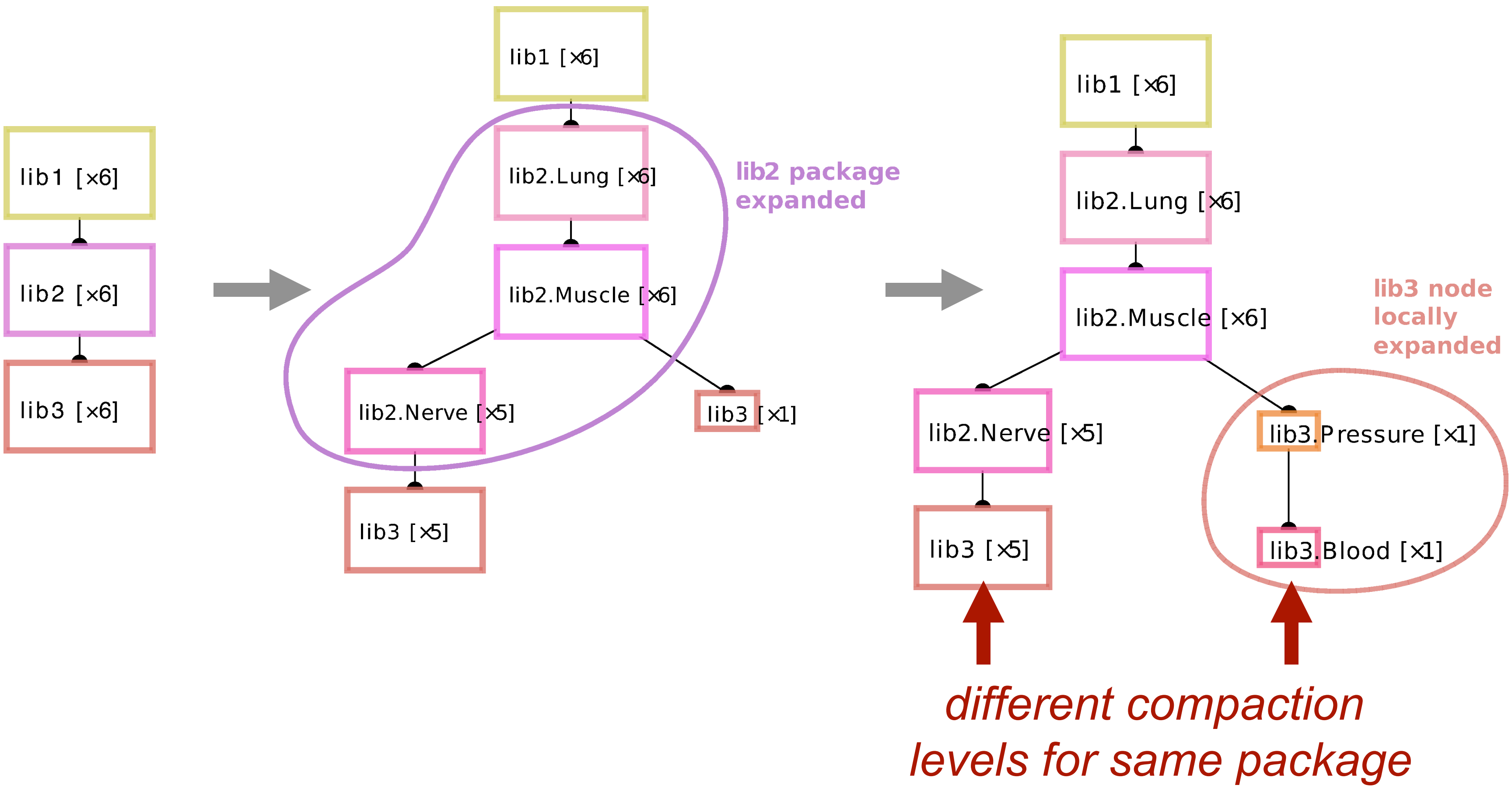
- Only highest level packages visible



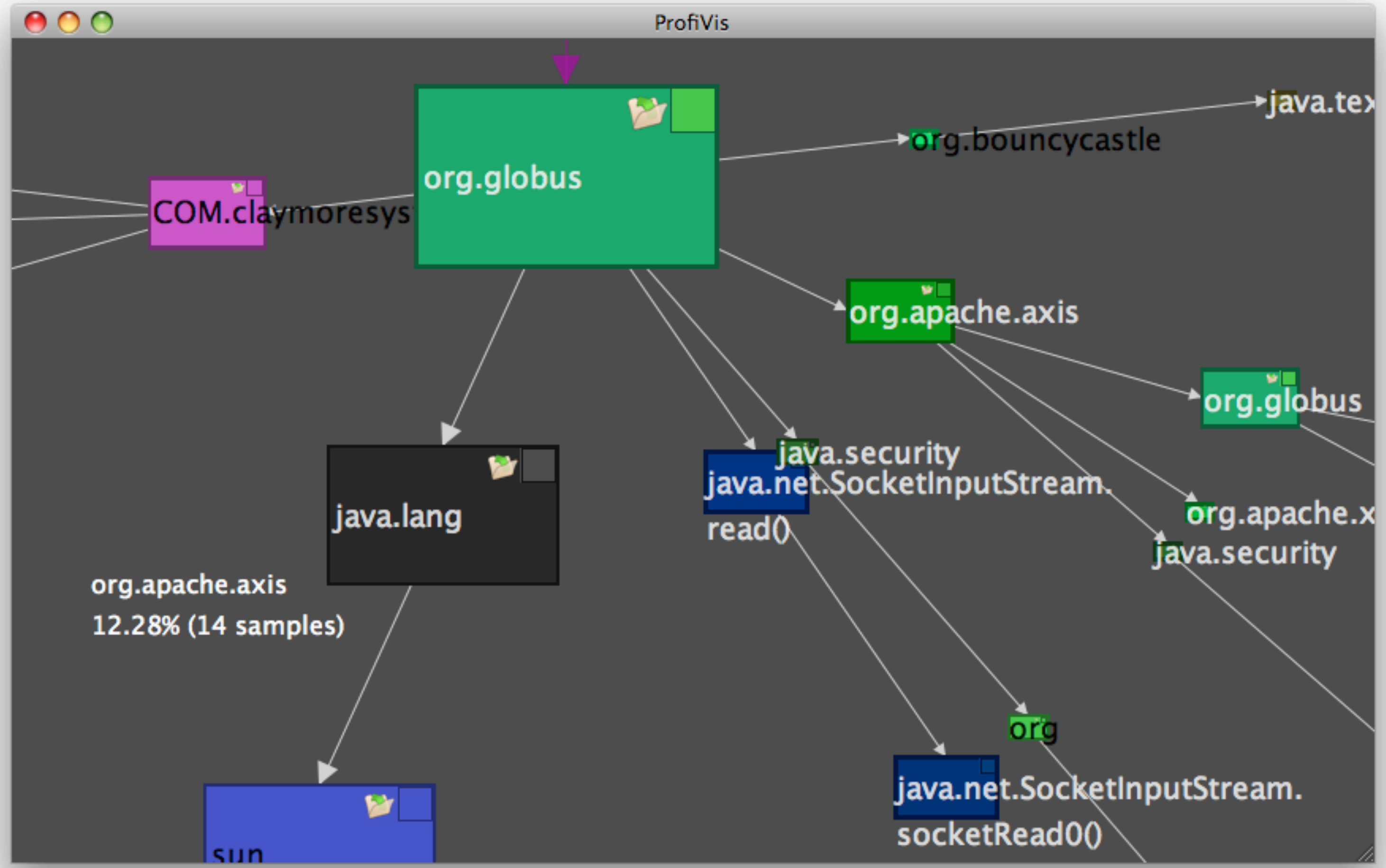


Progressive exploration

- Different levels of compaction in different parts of graph
  - ➔ including for the same package (here lib3)



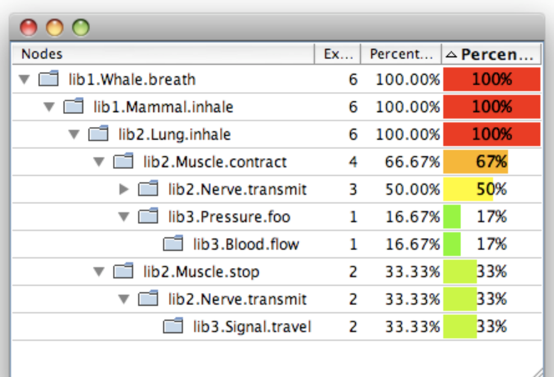
Demo



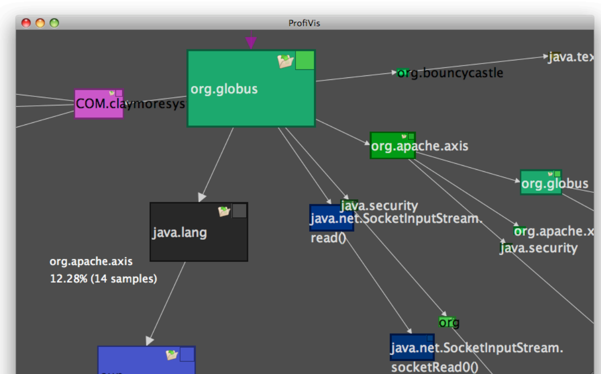
ProfiVis
java.tex
org.bouncycastle
org.globus
COM.claymoresys
org.apache.axis
org.globus
java.security
java.net.SocketInputStream.
read()
java.lang
org.apache.x
java.security
org.apache.axis
12.28% (14 samples)
org
java.net.SocketInputStream.
socketRead0()
sun

Evaluation

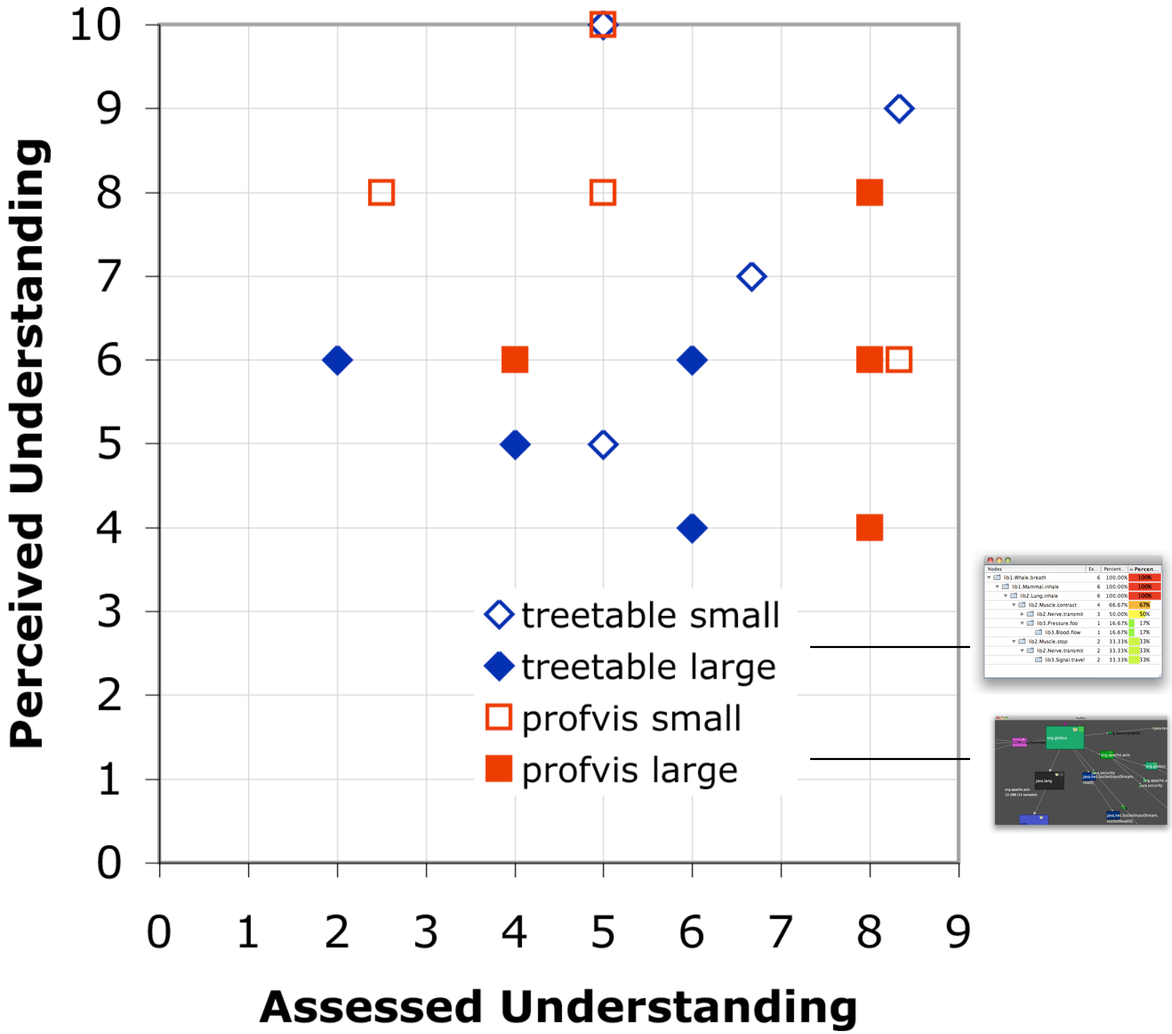
- **Goal**: explorative user study (4 users)
  - task for users: identify performance issues
  - 2 categories of programs (‘small’ and ‘large’)
  - Baseline: Textual Tree Table
- **Measures**
  - Perceived & assessed understanding
  - Interaction logs



Nodes	Ex...	Percent...	Percen...
lib1.Whale.breath	6	100.00%	100%
lib1.Mammal.inhale	6	100.00%	100%
lib2.Lung.inhale	6	100.00%	100%
lib2.Muscle.contract	4	66.67%	67%
lib2.Nerve.transmit	3	50.00%	50%
lib3.Pressure.foo	1	16.67%	17%
lib3.Blood.flow	1	16.67%	17%
lib2.Muscle.stop	2	33.33%	33%
lib2.Nerve.transmit	2	33.33%	33%
lib3.Signal.travel	2	33.33%	33%

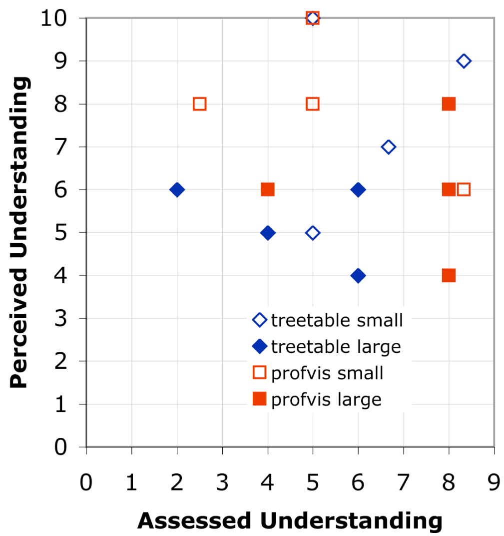


Results: Understanding



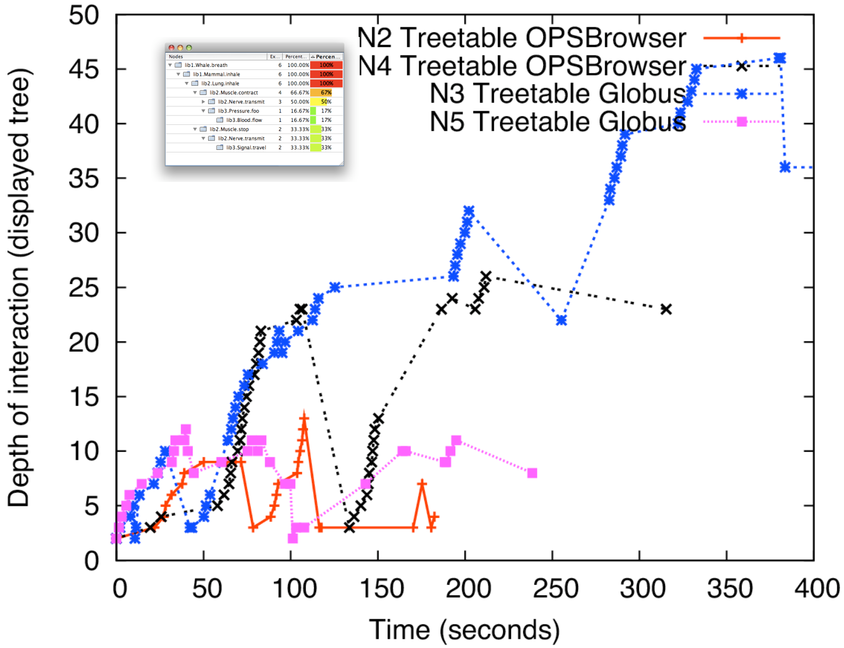
Findings

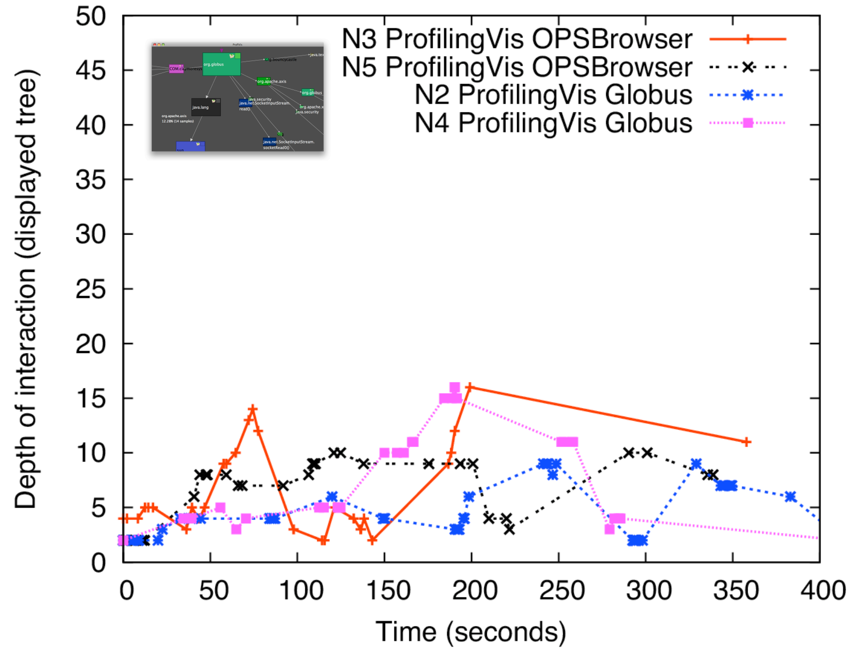
- Disconnection perceived/assessed on large programs
  - ➔ Users **overestimate** themselves with TreeTable
  - ➔ Users **underestimate** themselves with ProfVis
- Possible cause (?):
  - ➔ TreeTable hides full scope while ProfVis does not
  - ➔ **‘false sense of mastery’**



Results: Interaction

- Usage patterns seem to support this interpretation
  - ➔ users go deep w/ TreeTable, tend to hover w/ ProfVis





Summary

- High **reuse** can come with **drawbacks**

- But existing **abstractions** can help
- **Impact** of this line of research
  - ➔ Interdisciplinary links created with Psychology Dept.
  - ➔ Publications: SP&E, IEEE HPDC, ACM SoftVis
  - ➔ Talks and videos: AT&T, IBM, Cambridge, YouTube
  - ➔ Tool available on-line: http://ftaiani.ouvaton.org/7-software/profvis.html
  - ➔ Already used at Lancaster & IRISA

Outline

- **Intro** (just done)
- **WhisperKit:**
  Programming Gossip-based Systems

- **ProfVis:**
  Anomaly Diagnosis in Grid Middleware

- **Conclusion and Outlook**

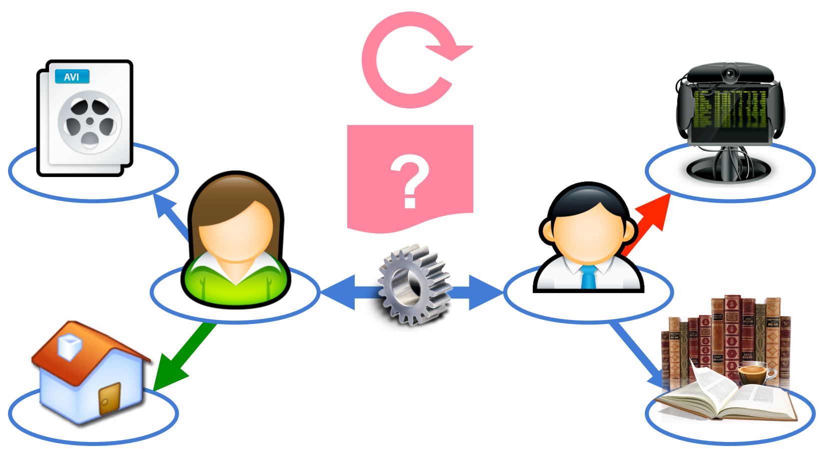
Conclusion

- **Reuse** and **abstraction** in **2 large-scale dist. systems**
  - ➔ 2 contributions: WhisperKit, Profvis
  - ➔ in 2 representative systems: gossip, grid
  - ➔ both proposal (mechanisms, abstractions) & study (tools)

- Emerging messages
  - ➔ **feasible** and **beneficial** (GossipKit)
  - ➔ but **own challenges**, that must be studied
  - ➔ by **reconsidering** some soft. eng. techniques (CBSE/DSL)
  - ➔ by **studying** existing production systems (Globus/CORBA)

Outlook: Social Networks

- Rapidly emerging
  - ➔ 800M Facebook users, 10M foursquare users
- How best to **program** fully decentralised versions?
  - ➔ Different mechanisms needed in different parts of networks
  - ➔ Different mechanisms for different features
- How to support **Adaptation** / **Composition** / **Synergies**?



The End

(Thank you)