Small and simple middleware for truly large distributed systems

Maarten van Steen

vrije Universiteit  amsterdam
• To hosts: thank you for inviting me to give a presentation at LADIS
• Of course, question comes to mind: what should I talk about, although it’s obvious that I’ve been working on large-scale distributed systems/middleware.
• After some thought, I decided to share some ideas on how to build SMALL middleware for LARGE systems.
Background

• We have been looking at large-scale distributed systems for quite some time now

• Most, if not all, our designs and implementations are (too) complex
  – Httpd: 230 kLOC
  – Globule: 23 kLOC
  – Globe: 218 kLOC
  – JBoss server: 949 kLOC
  – CORBA (mico): 189 kLOC

• We are now taking a deliberate choice to reconsider how we should build large-scale systems while sticking to simplicity
Overview

- Distribution transparency revisited:
  - Transparency seems to be the cause of many complex solutions
  - Devised a unifying messaging system based on local-only executions

- Debugging support for very large wireless sensor networks
  - Cannot have the nodes continuously monitor everything
  - Need flexible, generic, yet application-specific support
  - Severely resource-constrained
In this talk, I'd like to stand still at two projects we are currently carrying out in which simplicity is a key issue.

In the first, and major part of this talk, I'd like to take you along an avenue where we revisited distribution transparency.

The second part covers network debugging for very large WSNs.

For both cases: we are not there yet, but forcing ourselves back to the drawing boards is shaping our thoughts on simplicity for scalable solutions.
Distribution transparency revisited

Observation: Middleware solutions aim at achieving distribution transparency

hide the fact that we are dealing with distributed processes, data, and control

However: transparency can never be completely achieved (impossibility results) and can easily lead to false expectations
Distribution transparency revisited

- fundamentally different semantics between local and remote objects\(^1\)
- local method invocations behave differently from remote method invocations in the presence of failures\(^2\)
- replicated data often does not adhere to well-understood strong consistency models, but only to weaker models.
(1) Compare e.g., copy semantics in method invocations: local objects are copied deep; remote objects are shallow-copied.

(2) RMI s need additional code to handle communication failures.

(3) Think of Web-caching policies, or the intricacies of interactions between mail clients and servers. Even worse: consistency in parallel programs.
Distribution transparency revisited

Conclusion: We need to rethink about distribution transparency when developing middleware.

Special case: Remote invocations:

- **Aim:** Hide differences with local invocations
- **Problem:** Remote invocations have very different failure semantics in comparison to local invocations

Can we develop a middleware system with no remote invocations, yet which avoids the intricacies of having only message-passing?
• We decided to concentrate on the most obvious one: RMIs
• As said, these have very different failure semantics compared to LMI
Objects as messages

Main design principle: all operations are carried out locally; if needed, data/objects are first copied to operator’s location.

Motivation: Execution of local operations is clearly understood, also in the presence of failures.

copy-before-use principle
• Main issue: We wanted to have objects as messages, and allow only local operations on objects.
• Note that the semantics of most local operations are clearly understood.
• Executing an operation can never be done by a remote process: copy-before-use.
Micro objects

<table>
<thead>
<tr>
<th>Distributed</th>
<th>Not Distributed</th>
</tr>
</thead>
<tbody>
<tr>
<td>token</td>
<td>payload security</td>
</tr>
<tr>
<td>expire date</td>
<td>policy</td>
</tr>
<tr>
<td>home location</td>
<td>data</td>
</tr>
<tr>
<td>hash</td>
<td></td>
</tr>
<tr>
<td>payload</td>
<td>cluster security</td>
</tr>
<tr>
<td>application data</td>
<td>policy</td>
</tr>
<tr>
<td>cluster</td>
<td>data</td>
</tr>
<tr>
<td>token 1</td>
<td>cluster replication</td>
</tr>
<tr>
<td>...</td>
<td>policy</td>
</tr>
<tr>
<td>token n</td>
<td>data</td>
</tr>
</tbody>
</table>

immutable

mutable

closed shared

shared
• This lead us to the concept of **micro objects**
• Principle: placeholder for some (immutable) data, along with an **append-only** set to refer to other micro objects.
• Important: referenced micro objects may be located somewhere else ⇒ **distributed data structures**, notably **distributed graphs of micro objects**.
• Let's have a closer look at these type of objects.
Micro objects

- token
  - expire date
  - home location
  - hash

- payload
  - application data

- cluster
  - token 1
  - \ldots
  - token n

- payload security
  - policy
  - data

- cluster security
  - policy
  - data

- cluster replication
  - policy
  - data

- distributed
- immutable
- mutable
- not distributed
• Every micro object has an associated token that can be used as a unique identifier
• Every object has an associated home server from where the payload can be retrieved (guaranteed) until the object expires.
• We do not support a delete operation; instead, every micro object has an associated expiration time that tells how long the home server will host the object
• The hash is computed over the location, expire time, and payload
• Note: the token is immutable, and can be freely distributed across a system
Micro objects

- Token
- Expire date
- Home location
- Hash
- Token
- Application data
- Cluster
- Payload security
- Policy data
- Cluster security
- Policy data
- Cluster replication
- Policy data

Distributed

- Immutable
- Closed
- Shared

Not distributed

- Mutable
- Open
- Private
• The same holds for the payload: this is immutable data.
• We’ll see an example later how data updates are actually handled.
• In principle, the payload is encrypted.
Micro objects

- Token
  - Expire date
  - Home location
  - Hash
- Payload
  - Application data
- Cluster
  - Token 1
  - ... token n
- Distributed
- Not distributed
  - Payload security
    - Policy data
  - Cluster security
    - Policy data
  - Cluster replication
    - Policy data

- Immutable
  - Token
  - Home location
  - Hash
- Mutable
  - Cluster

- Closed shared
- Open shared
• Clusters are comparable to append-only lists: tokens (references to objects) can only be added to a cluster, but never removed unless the referenced object no longer exists.

• **Note:** A cluster forms the building block for arbitrarily complex distributed data structures.

• **Also note:** clusters are mutable, and fully distributable throughout the network.
Micro objects

- token
  - expire date
  - home location
  - hash
- payload
  - application data
- cluster
  - token 1
  - \ldots
  - token n

Distributed
- token
- payload
- cluster

Not distributed
- payload security
  - policy
  - data
- cluster security
  - policy
  - data
- cluster replication
  - policy
  - data

Immutable
- token
- payload
- cluster

Mutable
- token
- payload
- cluster

Closed shared
- token
- payload
- cluster

Shared
- token
- payload
- cluster
Let’s now have a look at the nondistributed part of a micro object. Payload is protected through encryption. Policies and keys are nondistributed and disclosed only within a specific group. Likewise, cluster membership may be encrypted as well and disclosed only to authorized servers.
Micro objects

- **distributed**
  - token
  - expire date
  - home location
  - hash
  - payload
  - application data
  - cluster
    - token 1
    - \(\vdots\)
    - token n

- **not distributed**
  - payload security
    - policy
    - data
  - cluster security
    - policy
    - data
  - cluster replication
    - policy
    - data

- **immutable**
  - closed
  - shared

- **mutable**
  - open
  - shared
• Finally, the way clusters are replicated (i.e., the references to tokens associated with a micro object), is shared with other parties, but is not distributed throughout the network.

• **Important:** by not distributing cluster replication policies, we effectively establish that every peer/host holding a copy of a micro object can decide whether and how to participate in a replication scheme.
Micro objects in action

Scenario:

- Alice, Bob, and Chuck want to exchange news items.
- Alice starts with creating a micro object $M$ for storing news items.

Alice’s server becomes the home for $M$

- $M$ is passed to Bob and Clare.
Micro objects in action

Bob creates $N_1$ and floods to Alice
• **Situation:** Alice, Bob, and Clare each have a copy of $M$. They all decide to abide by the rules of flooding news items.

• **Principle:** news items are appended to $M$’s cluster, after which they are flooded to known peers.

• **Example:** Bob and Clare both know about Alice. When Bob creates $N_1$, it appends it to the cluster of its local copy of $M$. Bob’s server will then send a copy of $N_1$ to Alice.
Micro objects in action

Alice receives $N_1$; flooding stops
- **Situation:** Alice and Bob do not know about $N_1$, but Clare does not.
Micro objects in action

Clare creates $N_2$ and floods to Alice
• **Situation:** Clare creates $N_2$, and subsequently sends a copy to Alice.
Micro objects in action

Alice floods $N_1$, $N_2$ to Bob and Clare
• Alice receives not only $N_2$, but continues flooding to other known peers, in this case Bob.
• Alice also discovers Clare, and will attempt to flood $N_1$ to her.
• **Note:** there can be various (efficient) implementations of flooding algorithms.
Micro objects in action

Flooding stops
• End result: Alice, Bob, and Clare know all about news items $N_1$ and $N_2$ as well as the peers holding copies of $M$. 
A few observations

- A micro object’s payload is passed from $A$ to $B$ only if both servers have matching replication policies.\(^\text{1}\)
- Recipients are in control: data cannot be unwillingly pushed toward them.
- Extremely simple update model:
  - Payload is immutable
  - Tokens can only be added to clusters
- Updates effectively happen only on distributed data structures
(1) Matching does not necessarily imply the same policy. It could also be that Alice merely specifies to accept objects that are flooded by Clare.
A more advanced example

We can construct a shared distributed file $F$:

- The file itself is represented by a micro object $F$.
- Each block $B_i$ is represented by a micro object $B_i$.
- The content of block $B_i$ is represented by a micro object $C_i[t]$, ordered by expiration date (i.e., $C_i[t]$ is more recent than $C_i[t']$ iff $t \geq t'$).

Note: We are assuming (loosely) synchronized clocks in order to reflect a notion of most recent update.
A more advanced example

Distributed application object

F
B1
-
3
4
2
1
0
B2
C
- - -
file (F)
content (C)
block (B)
Replication

Observation: replication comes very natural with micro objects when introducing levels. Consider a micro object $M$:

- Level 0: only the tokens in the cluster associated with $M$ are subject to replication.
- Level 1: the micro objects associated with those tokens will be cached and replicated.
- Level 2: the tokens in the respective clusters of those replicated objects...
Replicated file

- Level 0: add $C_2[3] \Rightarrow$ nothing happens (no new blocks)
- Level 1: add $C_2[3] \Rightarrow$ nothing happens (no new blocks)
- Level 2: add $C_2[3] \Rightarrow$ its token is replicated (aka: notification)
- Level 3: add $C_2[3] \Rightarrow$ content block is replicated
Design issues

- **Identifying data**: All data is contained in a micro object with a systemwide unique ID.

- **Locating data**: Every micro object has an associated home server responsible for keeping that object available until it expires. **Note**: availability can be increased through standard, well-known techniques.

- **Deleting data**: No explicit delete, only by expiration. This choice eliminates a lot of synchronization! **Note**: an object’s lifetime can be extended by any server willing to do so.
Design issues

- **Updating data:** Data is never really updated, but instead (eventually) replaced by other objects. **Note:** Clusters can only be extended with tokens, whereas only tokens of expired objects may be removed.

- **Protecting data:** Payloads and cluster membership are encrypted and disclosed only to authorized servers. **Note:** we have no special measures against traffic analysis.

- **Replicating data:** The cluster of each (copy of a) micro object has its own associated replication policy, determined by the server hosting that copy. Micro objects can be copied between servers only if the replication policies of their respective associated clusters match.
Example: Unified Messaging

Observation: There is no fundamental difference between various (user level) messaging systems, yet there is a myriad of interfaces and protocols.

Question: Can we build a system that unifies the various approaches?
Current Messaging
Unified messaging basics

**TISM:** Targetable
**Immutable Short Message:** the micro object that holds the data associated with a message.

**Target:** a distributed object for collecting TISMs.
• Presence DAO: a ping DAO is a short-lived micro object notifying that a specific user is now in the system. In effect, a message is sent to all interested.
TISM design
• **body**: most important part, consisting of the actual parts that make up the message.
• **attachments**: collection of attachements, each consisting of their own parts.
• **TISM tree**: allows to build threads of messages
TISM replication

Diagram showing the structure of TISM replication with levels 1 to 6.
Playing with keys

Problem: How can we protect unauthorized access to TISMs contained in a target \(\Rightarrow\) every target has an associated \((\text{post-key}, \text{read-key})\)-pair:

- **Post-key:** When holding the post-key associated with a target, the sending MO server is allowed to post a TISM in the target.

- **Read-key:** The holding MO server is allowed to read TISMs from a target.
Messaging models
Say-all, hear-all
Messing models
Moderated mailing list

moderated
moderator
unmoderated
target

moderated
target

unmoderated
target

moderator
Messaging models
Fully controlled mailing lists

Automated forwarding

A → B → C → D → T → A
UMS status

- **Prototype** implementation available, suitable for various messaging services
- Working on **GAIM plugin** to unify e-mail, IM, and news services
- **Strong:** Highly flexible model, with clear operational semantics (everything is local)
- **Strong:** Highly scalable, allowing for efficient implementations
- **Weak:** Unclear how portable to other platforms (e.g., wireless ad hoc networks)
Debugging Wireless (Sensor) Networks

Starting point: We expect that very large wireless sensor/actuator networks will be deployed in the near future:

- Systems may consist of (tens of) thousands cheap wireless networking nodes.
- Examples: when “cheap” deployment is an issue:
  - (Ad hoc) monitoring of pollution at airfields
  - Checking for failures of streetlights
  - Asset control (groups of items that should stay together)
Network debugging

Problem: Is there an effective way to monitor and debug the network while taking resource utilization into account:

- A node should not have to pessimistically monitor its environment as this may rapidly take up all its energy resources.
- Continuous monitoring also requires CPU and communication cycles that may be needed for real work.

Also: it is virtually impossible to anticipate the problems that will be encountered.
Case study: MyriaNed network

- CPU: Atmel AVR (ATMega645)
- 2Kb EEPROM, 4Kb SRAM, 64Kb Flash (all on-chip)
- Radio: Nordic nRF24L01 (2.4GHz ISM, 125 chan., 2 Mbps)
- Packet: 3-byte address, 32-byte payload, 2-byte CRC
- First 500 nodes to be delivered this month
- Plan: build networks with up to 10,000 nodes
Design issues

• We need to be able to decide on-the-fly which application variables need to be monitored.

• We should allow for local coordination between several nodes (think of clock synchronization).

• We cannot afford continuous reporting of values over the network ⇒ control when debugging should happen.

• Communication is expensive ⇒ support local problem diagnosis as much as possible.
Example scenario

Base station suspects faults in part of the network
Example scenario

Neighboring nodes are put into monitoring mode
Example scenario

Nodes report back to base station on status
Dynamically extending an application

Problem:

• We want a general-purpose infrastructure for inspecting application variables
• Debugging code is application-specific ⇒ dynamic application extensions
• Inspecting application variables should be as nonintrusive as possible

Solution: Combine a virtual machine for debugging with the actual application, of which the binary has been extended with (possibly void) breakpoints.
Dynamically extending an application
Dynamically extending an application

APPLICATION

SOME FUNCTION

VM

SCRIPT i

[breakpoint handler]

active breakpoint

I/O finished

post I/O operation

I/O operation
Accessing application state

Problem: Debugging scripts need to access (and possibly modify) application state,

- we don’t want to write applications with these scripts in mind
- we don’t want to anticipate on which state needs to be accessed

Solution: Take application binaries and create map of (statically allocated) global variables. Note: static allocation of critical data is commonly applied in embedded systems.
Accessing application state

- **Application Sources** → compile → **Application Binaries**
- **Application Map**:
  - extract variable types
  - extract variable locations
  - use the map for appl. variable references
- **Debugging Script Sources** → translate → **Debugging Script Bytecode**
Inserting breakpoints

Problem: As we do not know in advance where to break into an application, there are two options:

- Dynamically insert breakpoints ⇒ computationally infeasible
- Pessimistically insert breakpoints at compile time, most of which will have void handlers

Note: Void breakpoints take 3 cycles to complete, introducing an overhead of a few percent, depending on the application.
So far

• Very preliminary prototype implementation available for TinyOS environment

• Working on flexible solution for self-organization of nodes into hierarchical organization (needed for routing and other things)

• Squeezing solutions into currently available memory may be very difficult

• In parallel: investigating self-healing capabilities (let nodes decide on what to debug)
Conclusions

• Simplicity turns out be surprisingly challenging
• Unclear to what extent we can actually achieve our goals (notably wrt WSNs)
• Exploration of well-known techniques is promising:
  – gossiping
  – local-only decision making
• But also: we need to think big in terms of network sizes
Credits

Jan-Mark Wams   Konrad Iwanicki

Questions?