Development of dynamically evolving and self-adaptive software

4. Dynamic software update

LASER 2013
Isola d’Elba, September 2013

Carlo Ghezzi
Politecnico di Milano
Deep-SE Group @ DEIB
The problem

• Once you detect that a change has to be made to your running configuration, how do you react?

A solution

• Replace distributed configuration A with B

Assumption

• Componentized distributed implementation
• Implementation structured as a program family
• Reconfiguration = new family member instantiated
The problem

• Traditional approach to software updates
  - Shut-down the system
  - Perform update
  - Restart
• Traditional program families instantiated statically
• Here we need a dynamic program family
Requirement

• The dynamic update has to be:
  • **Efficient**
    • Low disruption
      *minimize the interruption of the system’s service*
    • Timely
      *minimize the delay with which the system is updated*
  • **Safe**
    • It must not lead the system into unexpected erroneous behavior.
Our setting

- We assume we know the architecture of the system
  - System components and their interaction
- Based on the architecture, we define how can we safely replace a component at run-time by
  - providing a **condition** to safely and efficiently substitute components at run-time
  - an **algorithm** to ensure this condition
  - a **framework** that practically enables safe run-time replacement of components when this condition is satisfied
Software architecture model

- Component-based distributed systems may be described by the **static configuration**:
  - A directed graph whose:
    - *nodes* represent **components** with in-ports and out-ports
    - *directed edges* represents **static dependences**

![Diagram of a directed graph with labeled nodes: Portal, Auth, DB, Proc, and labeled edges representing static dependences. The diagram includes the static configuration of an example CBDS.]
Transactions

• A transaction is a sequence of actions executed by a component that completes in bounded time
  • Actions include:
    • Local computations
    • Message exchanges
  • A transaction can be:
    • a root transaction if initiated by an outside client
    • a sub-transaction if initiated by another transaction
  • A distributed transaction includes the root transaction and all (direct and indirect) sub-transactions
Transactions

getToken (cred)
return token
process (token, data)
verify(token)
ok

t0

root transaction

T1

sub-transaction

T3

T0

T2
dbOperation()
db result

T4

proc result
Requirement for a safe replacement a component at run-time

- We assume that the corresponding off-line update is correct
- The replacement is safe (correct) if
  - The transactions that end before the update satisfy the old specification $S$
  - The transactions that begin after the update satisfy the new specification $S'$
  - The transactions that begin before the update, and end after it, satisfy either $S$ or $S'$
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept **version consistent**
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept version consistent
Replacing Auth when idle

Portal

Auth

gToken(cred)

T0

return token

T1

process(token, data)

Proc

verify(token)

T3

OK

DB

dbOp()

T2

T4

Idle:
It is not currently hosting transactions.

B

Idle:
It is not currently hosting transactions.
Replacing Auth when idle

Idle:
It is not currently hosting transactions.

getToken(cred)

return token

process(token, data)

verify(token)

OK

dbOp()
Replacing Auth when idle

Portal

getToken(cred)

return token

Auth

process(token, data)

verify(token)

OK

Proc

dbOp()

DB

T0

T1

T2

T3

T4

Unsafe!

C

✔
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept **version consistent**
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept **version consistent**
State of the art: Quiescence

A node is **quiescent** if it is idle and the rest of the world has nothing to do with it.

Precisely (Kramer and Magee):

1. it is not currently engaged in a transaction that it initiated
2. it will not initiate new transactions
3. it is not currently engaged in servicing a transaction
4. no transactions have been or will be initiated by other nodes which require service from this node.
Achieving quiescence

- To make a component quiescent, one need to passivate it and all components depending on it.
Achieving quiescence

- To make a component quiescent, one need to passivate it and all components depending on it.
Achieving quiescence

• To make a component quiescent, one need to passivate it and all components depending on it.
Achieving quiescence

• To make a component quiescent, one need to passivate it and all components depending on it.
Replacing Auth when quiescent

Portal

AuthToken(cred)

return token (T1)

internal process(token, data)

verify(token)

OK (T3)

T2

dbOp()

DB

T4
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept **version consistent**
How can we safely replace a component at run-time?

Different possible answers

1. when it is **idle** (not currently holding a transaction)
2. when it is **quiescent**
3. when it is **tranquil**
4. in such a way that all transactions are kept **version consistent**
State of the art: Tranquility

A node is **tranquil** if it is idle and it is not between two interactions with the same neighbor.

Precisely, (Vandewoude)

1. it is not currently engaged in a transaction that it initiated,
2. it will not initiate new transactions,
3. it is not actively processing a request, and
4. none of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.

Replacing Auth when tranquil

1. Portal
   - getToken(cred)
   - T0
   - return token
   - T1

2. Auth
   - process(token, data)
   - T2
   - verify(token)
   - T3
   - OK
   - T4

3. Proc
   - dbOp()
Replacing Auth when tranquil

getToken(cred)

process(token, data)

verify(token)

OK

dbOp()
Replacing Auth when tranquil

getToken(cred)

return token

process(token, data)

verify(token)

OK

dbOp()
Replacing Auth when tranquil

Portal

AuthToken(cred)

return token

token

process(token, data)

verify(token)

OK

dbOp()

DB

Proc

T0

T1

T2

T3

T4

A

C

D

✔

✔

✔

Sunday, September 15, 13
How can we safely replace a component at run-time?

- Different possible answers
  1. when it is **idle**  ← Not enough!
  2. when it is **quiescent**  ← Safe, but can be disruptive
  3. when it is **tranquil**  ← Less disruptive, but can be unsafe
How can we safely replace a component at run-time?

- Different possible answers
  1. when it is **idle** ← Not enough!
  2. when it is **quiescent** ← Safe, but can be disruptive
  3. when it is **tranquil** ← Less disruptive, but can be unsafe
  4. in such a way that all transactions are kept **version consistent**
How can we safely replace a component at run-time?

• Different possible answers
  1. when it is idle $\leftarrow$ Not enough!
  2. when it is quiescent $\leftarrow$ Safe, but can be disruptive
  3. when it is tranquil $\leftarrow$ Less disruptive, but can be unsafe
  4. in such a way that all transactions are kept version consistent
Version consistency

• A criterion for dynamic reconfiguration of CBDSs
  • as safe as the Quiescence
  • more timely and less disruptive
• We introduce a Distributed Management Algorithm to achieve VC for the component (or the part of the system) to be updated.
**Version consistency**

Intuitively:

any extant transaction, with all its (direct and indirect) sub-transactions, is executed as if it were entirely in the old or entirely in the new configuration.

*Transaction T is version consistent with respect to a runtime update of a set of components ω with a new version ω’* iff

\[ \neg \exists T_1, T_2 \in \text{ext}(T) \mid h_{T_1} \in \omega \land h_{T_2} \in \omega’ \]

A runtime update is version consistent iff all transactions are version consistent with respect to the update.

\[ \text{ext}(T) = \{x \mid x = T \vee \text{sub}^+(T, x)\} \]

\[ \text{sub}(T_1, T_2) \quad T_1 \text{ subtransaction of } T_2 \]
Are the updates version consistent?

Portal
	getToken(cred)

Auth
	return token

Proc
	process(token, data)
	verify(token)
	OK

DB
	dbOp()
Ensuring version consistency

• Distributed runtime model of dynamic dependences
• Freeness as a locally checkable condition
• Methods for achieving freeness
Dynamic dependences
Dynamic dependences

To serve transactions in $\text{ext}(T_0)$, Portal might initiate transactions on Auth in the future.
Dynamic dependences

To serve transactions in $\text{ext}(T_0)$, Portal might initiate transactions on Auth in the future.

Auth has hosted transactions in $\text{ext}(T_0)$ initiated by Portal in the past.
Dynamic dependences

To serve transactions in \( \text{ext}(T_0) \), Portal might initiate transactions on Auth in the future.

Auth has hosted transactions in \( \text{ext}(T_0) \) initiated by Portal in the past.

Auth is currently hosting transactions in \( \text{ext}(T_0) \).
Dynamic dependences

To serve transactions in $\text{ext}(T_0)$, Portal might initiate transactions on Auth in the future.

Auth has hosted transactions in $\text{ext}(T_0)$ initiated by Portal in the past.

Auth is currently hosting transactions in $\text{ext}(T_0)$.

It’s a distributed runtime model.
Valid configuration

• A configuration with dynamic dependencies is a distributed runtime model
  - Dynamic dependences are represented by future and past edges created and removed at runtime
  - Each node is in charge of edges leaving from it and aware of edges entering it

• A valid configuration must satisfy certain invariants on edges
Freeness

Freeness is a locally checkable condition

Given a configuration with dynamic dependence $G$, a component $C$ is said to be free of dependence with respect to a root transaction $T$ if there does not exist a pair of future/past edges labeled with $T$ arriving at $C$.

A component is said to be free in $G$ if it is free of dependence by all root transactions whose identifiers appear in $G$.

Freeness is sufficient for version consistency

A dynamic update of a component is version consistent if it happens when the component is idle and free.
Managing dynamic dependences

The configuration with dynamic dependences is maintained at runtime, in a distributed way.

When a root transaction is started, a setup of all future edges is performed.

In practice, future edge information is stored locally.

- Each node stores info about its incoming past and future edges.
- Future edges are then eliminated and past edges created as the transaction is in progress.
- A cleanup phase that eliminates remaining future/past edges is performed upon termination of the transaction.
1) Setup (at the beginning of a rootTx):
Recursively create enough future edges
1) Setup (at the beginning of a rootTx):
   Recursively create enough future edges
1) Setup (at the beginning of a rootTx):
Recurisvely create enough future edges
1) Setup (at the beginning of a rootTx):
Recursively create enough future edges
1) Setup (at the beginning of a rootTx):
   Recursively create enough future edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):

Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):

Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
   Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
Progressively Remove Future Edges And Create Past Edges
2) Progressing (on the beginning/ending subTxs):
Progressively Remove Future Edges And Create Past Edges
3) Clean-up (on the ending of RootTx):
   Recursively remove past edges and remaining future edges
3) Clean-up (on the ending of RootTx):
   Recursively remove past edges and remaining future edges
Achieving freeness

1. Waiting for freeness to manifest itself (WF)
   • Low disruption, but no guarantee for timeliness. Freeness can be never achieved.

2. Using concurrent versions of the components (CV)
   • Our framework makes it easy to decide
     • Which version for which request, and
     • When the old version can retire.
   • Preferred If feasible. Timely and low disruptive.

3. Blocking transactions to achieve freeness (BF)
   • Timely, but more disruptive than WF/CV.
Evaluation

• **Goal:** To evaluate timeliness and disruption
  - Compare our approach with the quiescence approach
  - Study the impact of network latency

• **Method:** Discrete event simulations
  - Timeliness: time span between request and ready state
  - Disruption: loss of working time wrt. no reconfiguration
Experiment 1: Timeliness

Our approach can be (~20%) more timely
Experiment 1: Disruption

Our approach can be significantly less disruptive (~50% for BF and negligible for CV)
Experiment 2: Network Latency

Our approach is consistently better in disruption but its gain in timeliness diminishes while message delay increases.
From components to a specification-based approach


[SEAMS 2012] C. Ghezzi, J. Greenyer; V. Panzica La Manna, "Synthesizing Dynamically Updating Controllers from Changes in Scenario-based Specifications"

Acknowledgements

• Most of the ideas discussed here matured or developed thanks to a funding from the European Research Council (Advanced Grant IDEAS-ERC, Project 227977---SMScom)

• ...and thanks to