Dissemination of reconfiguration policies on mesh networks

François Fouquet, Erwan Daubert, Noël Plouzeau, Olivier Barais, Johann Bourcier, Jean-Marc Jézéquel

Dissemination of reconfiguration policies on mesh networks

François Fouquet, Erwan Daubert, Noël Plouzeau,
Olivier Barais, Johann Bourcier, and Jean-Marc Jézéquel
University of Rennes 1, IRISA, INRIA Centre Rennes
Campus de Beaulieu, 35042 Rennes, France
{Firstname.Name}@inria.fr

Abstract. Component-based platforms are widely used to develop and deploy distributed pervasive system that exhibit a high degree of dynamicity, concurrency, distribution, heterogeneity, and volatility. This paper deals with the problem of ensuring safe yet efficient dynamic adaptation in a distributed and volatile environment. Most current platforms provide capabilities for dynamic local adaptation to adapt these systems to their evolving execution context, but are still limited in their ability to handle distributed adaptations. Thus, a remaining challenge is to safely propagate reconfiguration policies of component-based systems to ensure consistency of the architecture configuration models over a dynamic and distributed system. In this paper we implement a specific algorithm relying on the *models at runtime* paradigm to manage platform independent models of the current system architecture and its deployed configuration, and to propagate reconfiguration policies. We evaluate a combination of gossip-based algorithms and vector clock techniques that are able to propagate these policies safely in order to preserve consistency of architecture configuration models among all computation nodes of the system. This evaluation is done with a test-bed system running on a large size grid network.

1 Introduction

Nowadays, the increasing use of Internet of Things devices for computer supported cooperative work leads to large systems. As these devices use multiple mobile networks, these systems must deal with concurrency, distribution, and volatility issues. This volatility requires dynamic auto-adaptation of the system architecture, in order to provide domain specific services continuously. Tactical information and decision support systems for on field emergency management are perfect examples of such highly dynamic systems. Indeed, these multi-user interactive systems built on mobile devices need frequent changes of architecture to deal with rapid system evolution (*e.g.* scale up or scale down of team, download of new software modules by the device user) or to cope with network disconnections. For such systems, the traditional design process "design, code, compile, test, deploy, use, iterate" does not work.

Dynamic adaptation, pursuing IBM's vision of autonomic computing, is a very active area since the late 1990's - early 2000's [9]. Modern component-based systems [15,4] provide a reflection and intercession layer to dynamically reconfigure a running system. But the reconfiguration process remains complex, unreliable and often irreversible in a volatile and distributed context. The use

of model-driven techniques for managing such run-time behavior (named models@runtime [3]) helps to handle software reconfiguration. Models@runtime basically pushes the idea of reflection [14] one step further by considering the reflection layer as a real model that can be uncoupled from the running architecture (e.g. for reasoning, validation, and simulation purposes) and later automatically resynchronized with its running instance to trigger reconfigurations. Kevoree is our open-source dynamic component model[1], which relies on models at runtime to properly support the dynamic reconfiguration of distributed systems. The model used at runtime reflects the global running architecture and the distributed topology. In Kevoree, when a distributed node receives a model update that reflects the target running architecture, the node extracts the reconfigurations that affect it and transform them into a set of platform reconfiguration primitives. Finally, it executes them and propagates the reflection model to other nodes as a new consistent architecture model.

In a highly distributed and volatile environment, one of the challenges is the propagation of reconfiguration policies. Handling concurrent updates of shared data is a second challenge to be solved, as two nodes can trigger concurrent reconfigurations. Consistent dissemination of models at runtime in distributed systems requires a synchronization layer that solves these two challenges: information dissemination and concurrent update. Research in the field of peer-to-peer communication has produced many algorithms to deal with information dissemination in a volatile context [6]. Many paradigms are available to deal with this concurrent data exchange problems (e.g. vector clocks [7]). In this paper, we adapt a combination of gossip-based algorithms and vector clocks techniques to safely propagate reconfiguration policies by preserving architecture models consistency between all computation nodes of a distributed system. We have implemented a specific algorithm, which propagates configuration changes in a consistent manner in spite of frequent node link failures, relying on its payload of configuration data to improve its efficiency. We provide qualitative and quantitative evaluations of this algorithm, to help answering the following questions: (i) What is the influence of communication strategy on the propagation delay of models? (ii) Does a high rate of node link failure prevent the propagation of models and what is the impact of link failures on propagation delays? (iii) Does the algorithm detect concurrent updates of models and does it handle reconciliation correctly?

The remainder of this paper is organized as follows. Section 2 presents the background of this work. Section 3 details the combination of a gossip-based algorithm and the vector clock techniques used to preserve architecture models consistency between all computation nodes of the system. Section 4 details our experiments to evaluate this combination. Section 5 discusses articles, ideas and experimental results related to our work. Finally, Section 6 concludes this paper and presents ongoing work.

2 Background

Kevoree is an open-source dynamic component model[1], which relies on models at runtime [3] to properly support the dynamic adaptation of distributed sys-

[1] `http://kevoree.org`

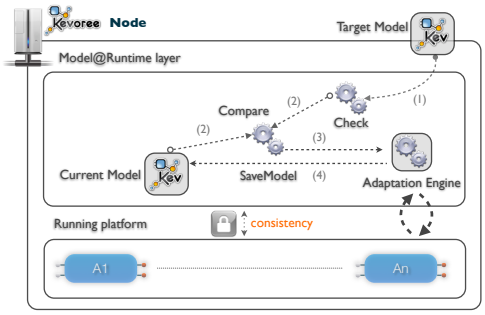

Fig. 1. Models@Runtime overview

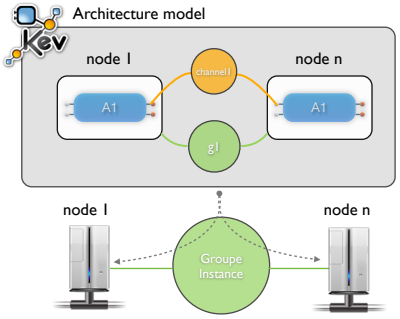

Fig. 2. Distributed reconfigurations

tems. Figure 1 presents a general overview of models@runtime. When changes appear as a new model (a target model) to apply on the system, it is checked and validated to ensure a well-formed system configuration. Then it will be compared with the current model that represents the running system. This comparison generates an adaptation model that contains the set of abstract primitives to go from the current model to the target one. Finally, the adaptation engine executes configuration actions to apply these abstract primitives. If an action fails, the adaptation engine rollbacks the configuration to ensure system consistency. Kevoree has been influenced by previous work that we carried out in the DiVA project [14]. With Kevoree we push our vision of models@runtime [14] farther. In particular, Kevoree supports distributed models@runtime properly. To this aim we introduce the *Node* concept in the model to represent the infrastructure topology. Kevoree includes a *Channel* concept to allow for multiple communication semantics between remote *Components* deployed on heterogeneous nodes. All Kevoree concepts (Component, Channel, Node) obey the Type Object pattern [8] to separate deployment artifacts from running artifacts. Kevoree supports multiple kinds of execution node technology (e.g. Java, Android, MiniCloud, FreeBSD, Arduino, ...).

Kevoree also introduces a dedicated concept named *Group*, to encapsulate platform synchronization algorithms. *Group* allows to define communication channels between nodes to propagate reconfiguration policies (i.e. new target model). This *Group* concept also encapsulates a dedicated protocol to ensure specific synchronization policies (e.g. Paxos derived algorithms for total order synchronization, Gossip derived algorithms for partial order and opportunistic synchronization). *Group*s can be bound to several nodes (named members), allowing them to explicitly define different synchronization strategies for the overall distributed system. This architecture organization is illustrated in Figure 2. In addition, a *Group* also defines a scope of synchronization, *i.e.* it defines which elements of the global model must be synchronized for the group's members. This avoids to globally share model@runtime models.

P2P algorithm and mesh network Schollmeier [16] defines a peer-to-peer network as "a distributed network architecture, where participants of the network share a part of their resources, which are accessible by the other peers directly, without passing intermediary entities". He also provides the following

distinction: *hybrid* peer-to-peer networks use a central entity, while *pure* peer-to-peer networks have no such entity. According to Wikipedia, a mesh network is "a type of network where each node must not only capture and disseminate its own data, but also serve as a relay for other nodes, that is, it must collaborate to propagate the data in the network". In these network topologies, gossip-like algorithms are good solutions to disseminate data.

Concurency data management for distributed message passing applications Distributed systems consist of a set of processes that cooperate to achieve a common goal. Processes communicate with data exchanges over the network, with no shared global memory. This leads to well known and difficult problems of causality and ordering of data exchanges. Solutions are known to cope with this problem: Lamport [10] defines an event order using logical clocks by adding a logical time on each message sent. Another solution was coinvented by Fidge [7] and Mattern [13], using a vector of logical clocks. In many cases the vector clock technique is the most appropriate solution to manage a partial order and concurrency between events[2], e.g. in distributed hash table systems such as Voldemort[2]).

Synthesis In our vision of distributed environments, system management is decentralized, allowing each peer to build, maintain and alter the overall architecture and platform models at runtime. Because of nodes volatility, ensuring consistency during reconfiguration is a critical task. We use Kevoree and the notion of *Group* to encapsulate platform synchronization algorithms with gossip and vector clock techniques.

3 An algorithm to disseminate reconfiguration policies

Each node holds a copy of the model that describes the overall system configuration. This system model contains a description of the nodes that currently compose the system, of components that are installed on each node and of network links between nodes. It also contains all information about *groups*. A *group* is the unit of model consistency for the models at runtime technique. Each node involved in model consistency includes several named *group instances*, which participates in the distributed model management for the local node. Part 1 of the algorithm provides the data definition for one node.

In addition to the information given by the model, each group instance maintains specific information (see algorithm's Part 2): a group id, a local copy of the model and the local node id. It also stores its current vector clock, a score for each of its neighbors and a boolean attribute to record whether the model has changed since the last time another node requested the local node's vector clock. The score of the neighbors is used to select the more interesting one when the local node looks for new reconfigurations.

[2] `http://project-voldemort.com`

Algorithm Part 1 DEFINITIONS

Message ASK_VECTORCLOCK, ASK_MODEL, NOTIFICATION
Type VectorClockEntry := <id: String, version ∈ ℕ>
Type Node // represents a node on the system
Type Model // represents a configuration of the system
Set Group:= {node: Node}
Set IDS(g: Group):= {id: String | ∃ node: Node, node ∈ g & node.name = id}
Set Neighbors(originator: Node, g:Group):= {node: Node | node ∈ g & originator ∈ g}
Set VectorClock(originator: Node, g: Group):= {entry: VectorClockEntry | entry.id == originator.name}
⋃ {entry1: VectorClockEntry |∃ node: Node, node != originator & entry1.id ∈ IDS(g) & node ∈ g}
Set VectorClocks(originator: Node, g: Group):= {vectorClock: VectorClock(originator, g)}

Main algorithm (see algorithm's Part 3). When a change appears on the model stored in a node, the corresponding group instance is notified. The group instance then sends notification to all its neighbors. These neighbors in turn may send a message to the current node, to ask for model update information. As the underlying communication network is volatile and unreliable, some notifications can be lost and not received by some members of a group. To deal with these losses, each member of a group asks periodically a chosen group member for changes. Since a model is a rather large data, group instances ask for the vector clock of the remote instance first, in order to decide if a model transfer is needed. More precisely, after comparing the vector clock received with its own vector clock, a group instance will request a model if both vector clocks are concurrent or if the vector clock received is more recent than its local one. Here concurrency means that each local and remote model have different changes which dot not appear on the other. A vector clock is more recent than another if some changes appear on it but not on the other. Upon reception of a model, the group instance compares the model's vector clock and the local clock again. If the local vector clock is older, the local node updates its local clock and also updates the local copy of the model using the model just received. If the vector clocks are concurrent then the group must resolve this concurrency at the model level to compute the correct model and then update the vector clock accordingly.

Functions SelectPeer (see Algorithm Part 4) In addition to this mechanism, each node pulls periodically one of its neighbors, in order to cope for lost notifications. The selection of the neighbor to pull is controlled by a score mechanism: a score is assigned to each peer by the group instance and the selection of the peer is done according to the smaller score. The score of the node grows when it is selected or when the network link to access this node seems to be down. The

Algorithm Part 2 STATE

g: Group ; changed: Boolean
currentModel: Model // local version of system configuration
localNode: Node // representation of local node
currentVectorClock ∈ VectorClocks(localNode, g)
scores := {<node: Node, score>, node ∈ Neighbors(localNode, g) && score ∈ ℕ}
nbFailure := {<node: Node, nbFail>, node ∈ Neighbors(localNode, g) && nbFail ∈ ℕ}

Algorithm Part 3 ALGORITHM

```
On init():
  vectorClock ← (localNode.name, 1)
  scores ← {Neighbors(localNode, g) × {0}}
  changed ← false
On change (currentModel):
  ∀ n, n ∈ Neighbors(localNode, g) → send (n, NOTIFICATION)
  changed ← true
Periodically do():
  node ← selectPeerUsingScore()
  send (node,ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode, g), NOTIFICATION):
  send (neighbor, ASK_VECTORCLOCK)
On receive (neighbor ∈ Neighbors(localNode,g), remoteVectorClock ∈ VectorClocks(neightbor, g)):

  result ← compareWithLocalVectorClock (remoteVectorClock)
  if result == BEFORE || result == CONCURRENTLY then
     send (neighbor, ASK_MODEL)
  end if
On receive (neightbor ∈ Neighbors(localNode,g),vectorClock ∈ Vectorclocks(neighbor, g), model)):

  result ← compareWithLocalVectorClock (targetVectorClock)
  if result == BEFORE then
     updateModel(model)
     mergeWithLocalVectorClock(vectorClock)
  else if result == CONCURRENTLY then
     resolveConcurrently(vectorClock, model)
  end if
On receive (neighbor ∈ Neighbors(localNode,g), request):
  if request == ASK_VECTORCLOCK then
     checkOrIncrementVectorClock()
     send (neighbor, currentVectorClock)
  end if
  if request == ASK_MODEL then
     checkOrIncrementVectorClock()
     send (neighbor, <currentVectorClock,currentmodel>)
  end if
```

down link detection relies on a synchronization layer. This layer uses model information to check for all available peers periodically and then to notify the group instance of unreachable nodes. A peer score takes into account the duration of unavailability of the peer. When the peer becomes available, this number is reset to 0: restored availability clears the failure record. Indeed, as the system uses a sporadic and volatile network, peers often appear and disappear and most of the time disappearance events are not causally connected.

Functions about vector clocks (see Algorithm Part 5) Our algorithm relies on vector clocks to detect changes in remote configuration models. When a local update of the model appears, a boolean called *changed* is set to true to ensure that upon a vector clock request from another node the group instance will increment by one its version id in its local vector clock before sending it to the requesting peer. In case of concurrent updates of models we rely on the use of the reflexivity provided by the model at runtime to solve the conflict. Priority is given to information about the nodes already reached and affected by the update. Any node detecting a conflict will merge these models and their associated vector clocks to store it as its current state. A reasoning upper layer

Algorithm Part 4 SelectPeer

```
Function selectPeerUsingScore()
  minScore := ∞ ; potentialPeers := {}
  for node → Neighbor(localNode, g) do
    if node ! = localNode && getScore(node) < minScore then
      minScore := getScore(node)
    end if
  end for
  for node → Neighbor(localNode, g) do
    if node ! = localNode && getScore(node) == minScore then
      potentialPeers := potentialPeers ⋃ {node}
    end if
  end for
  node := select randomly a node from potentialPeers
  updateScore(node)
  return node
Function getScore(node ∈ Neighbors(localNode, g))
  return scores(node)
Function updateScore(node ∈ Neighbors(localNode, g))
  oldScore := getScore(node)
  scores := scores ⋃ {node, oldScore + 2 * (nbFailure + 1)} \ {node, oldScore}
```

will then compute an update from this merged model by reading the model and correcting it. Description of this reasoning layer is beyond the scope of this paper and vector clocks merge and comparison is already defined on previous works on vector clocks [7] and [13].

Algorithm Part 5 FUNCTIONS

```
Function checkOrIncrementVectorClock()
  if changed == true then
    ∀ entry, entry ∈ currentVectorClock & entry.id == localNode.name ⇒ entry.v ← entry.v + 1
    changed ← false
  end if
Function compareWithLocalVectorClock(targetVectorClock∈VectorClocks(n ∈ neighbors(localNode, g), g)) // for details, please look at http://goo.gl/0tdEc
Function mergeWithLocalVectorClock(targetVectorClock ∈ VectorClocks(n ∈ neighbors(localNode, g), g)) // for details, please look at http://goo.gl/axbJN
Function resolveConcurrency(targetVectorClock ∈ VectorClocks(n ∈ neighbors(localNode, g), g))
  // for details, please look at http://goo.gl/bFTeH
Function updateModel(model ∈ Models)
  currentModel ← model
```

4 Evaluation

We have performed qualitative and quantitative evaluations of our algorithm, aiming at measuring the following indicators: (1) model propagation delay; (2) resilience to node link failure; (3) ability to detect concurrent models and to handle reconciliation. For each indicator we have set up an experimental protocol, using the firefighter tactical information case study metrics to simulate the system behaviour on a grid in different configurations. Although target platforms will be pervasive embedded systems, we have chosen a large scale grid as an evaluation testbed. The use of a grid allows us to stress the algorithm by setting up a large number of nodes but it also brings us more control over the parameters of the experiment e.g. network failure simulations. In this way experiments are reproducible, and reproducibility is essential to our experimental protocol. On-field validation is an ongoing work.

4.1 Common experimental protocol

Validation experiments share a common experimental protocol. Each experiment uses a set of logical Kevoree node deployed on physical nodes within a computer grid. Each Kevoree logical node is instantiated in a separate Java Virtual Machine and use the reference Kevoree implementation for JavaSE. The experimental grid is an heterogeneous grid that contains nodes of mixed computational power and type. Each node is connected to a local area network at 100 MB/s.

Topology model All our experiments take a bootstrap model as input, which describes the current abstract architecture (*i.e.* in its platform independent form). This abstract model contains information on node representations, node logical links and node communication group instances and relationships. This node set and these relationships describe a topology of the system, which is used by our synchronization algorithm. In order to improve the simulation of a firefighter tactical information case study, we use a random generator to create topology models that are organized in a cluster of clusters. In this way it is easier to simulate non-direct communication (*i.e.* node A cannot communicate directly with node B but must pass through node C).

Global time axis traces In order to track the propagation of new configurations in this distributed system, we decorate the algorithm with a logger. This logger sends a trace for each internal state change (*i.e.* new configuration or configuration reconciliation). These traces describe the current state of the group, namely the new vector clock, the identification of the peer originator of change and the network metrics used. In order to exploit temporal data on these traces without ensuring a global grid time synchronization we use a logger with a global time axis based on Java Greg Logger[3]. More precisely, this type of logger is based on a client server architecture. The server clock manages the global time reference. All clients periodically synchronize with the server, allowing it to store client latencies by taking into account clocks shift and network time transfer observed. Traces are emitted asynchronously by the client to the server, which then makes time reconciliation by adding the last value of latency observed for this client. All traces emitted by the server are therefore all time stamped accurately with the clock of the server. Finally, traces are chained by an algorithm to meet the following heuristic: a trace follows another one if it is the first occurrence that contains in its vector clock the originator node with its precise version number. Thus the final result for each experiment is a linked trace list on which we can precisely compute temporal results.

Communication modes We reuse mainly two classical exchange patterns to build our algorithm.
Pooling period term is associated with the time elapsed between two active synchronizations, and is initiated by a group member to another. In this synchronization step a vector clock and/or a model is sent back to the initiator.

[3] `http://code.google.com/p/greg/`

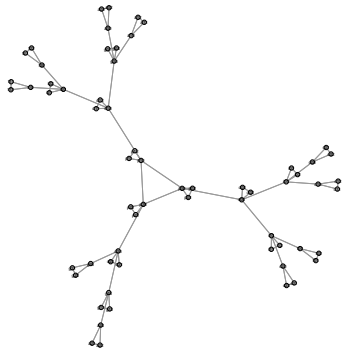

Fig. 3. Topology model of exp 1

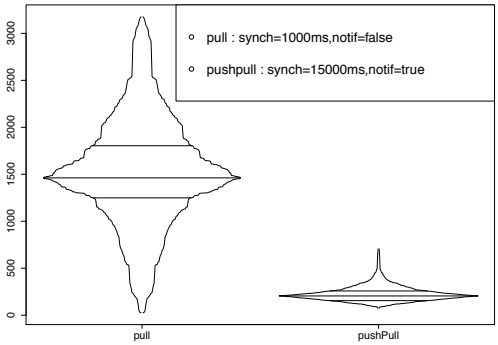

Fig. 4. Delay/hop(ms)

The Push/Pull technique is an association of the pooling active synchronization and an event-driven notification mechanism. This operation adds to the pooling mode a sending step to every reachable group member.

4.2 Experimental studies

Propagation delay versus network usage This first experiment aims at performing precise measurements of the capacity to disseminate model configurations. These measures will take care of the propagation delay and the network usage properties.

Experimental protocol As described in the common protocol subsection, measurements are performed on a computer grid. The probes injected in the Java implementation collect propagation delay and network occupation. After a bootstrap step on a topology model, a node chosen at random reconfigures its local model with a simple modification. In practice this reconfiguration step computes a new model, moving a component instance from one node to another chosen randomly. This new model is stored in the node, and the reconfiguration awaits propagation by the algorithm. This new configuration is tagged with the identification of reconfiguration originator. Figure 3 shows the topology model used for multi hop communication in the 66 nodes of this configuration. In this experiment, the network topology is static. No node joins or leaves the system.

The experiment is driven by the following parameters:(1) delay before starting an active check of the peers update (model synchronization);(2) activation of sending of change notification messages. To evaluate the impact of the second parameter, the experiment is run twice. In the first run, notifications are not used and the active synchronization delay is set to 1000 ms. In the second run, notifications are used and active synchronization delay is 15 s. In both cases, a reconfiguration is triggered every 20 seconds and each reconfiguration run takes 240 seconds, resulting in 12 reconfiguration steps.

Analysis The observed per hop propagations delays are presented as a percentile distribution (see Graph 4). The values displayed are the raw values of absolute

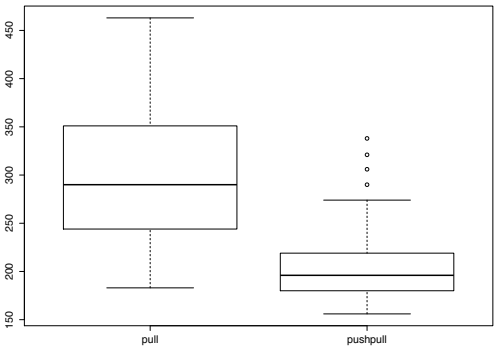

Fig. 5. Network usage/node(in kbytes)

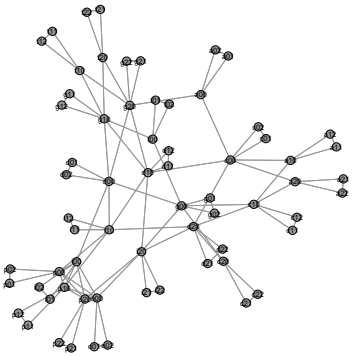

Fig. 6. Topology of exp 2

time logged divided by the minimum number of hops between the target and originator of the reconfiguration (the minimum being computed using a Bellman-Ford algorithm [5]). The traffic volume from protocol messages is shown in Figure 5 in KB per node per reconfiguration; the volume does not include payload. Absolute values of network consumption depends highly of implementation. Results presented here are from the Java version and can be vastly improved when targeting embedded devices like microcontrollers.

The use of notification reduces the propagation delay significantly: the average value decreases from 1510 ms/hop to 215 ms/hop. In addition, percentile distribution shows that the standard deviations of propagations are lower with in the version with notification. Thus this version of the algorithm has better scalability for large graph diameters.

However, in comparing the push pull and the push algorithm, the use of notification on network usage is not as significant. Analysis shows that these results are affected by cycles in the topology. When using notification of change, nodes in cycles will create parallel branches configuration diffusion. This in turn will increase the number of conflict resolution to be done, and these resolutions increase network consumption unnecessarily, by exchanging the same model version. When notifications are not used, pooling delays are large enough to avoid this concurrent configuration "flood". As the payload is a model with topology information, the notification algorithm could use this information to prevent flood. This solution will be studied in future work.

Failures impact on propagation delay A mobile mesh network such as the one used in a firefighter tactical information system is characterized by a large number of nodes that are often unreachable. We designed our algorithm to cope with these network problems. The second experiment described below tests the ability of the algorithm to disseminate new models in a mesh network with different failure rates.

Experiment protocol The experiment protocol is similar to the first experiment's one. The topology model is enhanced to provide a mesh network with many

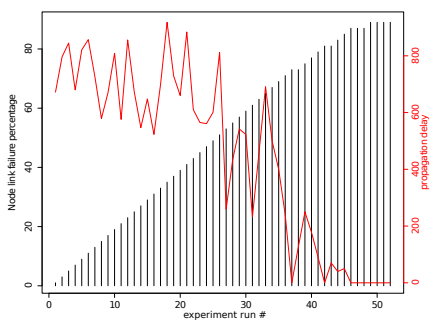

Fig. 7. Failure results

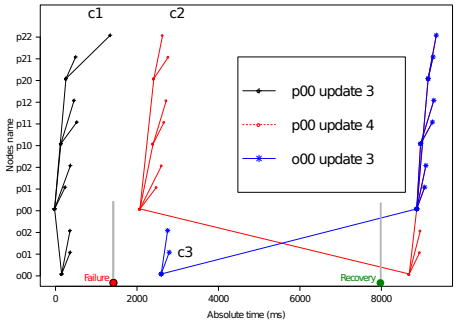

Fig. 8. Concurrent update

different routes between nodes (see Figure 6). At each run a modified model is pushed on a random node. The reconfiguration is similar to the previous experiment. During each run, additional failures are simulated on links between two nodes, according to a Poisson distribution. The failure rate is increased at each run, thus the number of reachable nodes decreases. To perform this failure simulation we inject probes, which also monitor synchronization events. At each run, the list of theoretically reachable nodes is computed and the initiator node waits for synchronization events from theses nodes. When all events have been received we compute the average propagation delay. In short, this experiment aims at checking that every theoretically reachable node receives the new configuration.

Analysis Figure 7 shows results of experiment #2. The histogram shows the rate of network failure for each run. The red curve displays the average propagation delay to reachable nodes (in milliseconds). Above a network failure of 85% the node originator of the reconfiguration is isolated from the network and therefore we stop the execution. With a failure rate under 85% every node receives the new configuration and we can compute the propagation delay.

Concurrency reconfiguration reconciliation Our third experiment addresses the problem of reconciliation and conflict detection between concurrent model updates. This problem occurs often in the firefighter tactical information case study architecture because of the sporadic communication capabilities of our network of nodes. As a node can stay isolated for some time, reconfiguration data no longer reaches it. Furthermore, local reconfigurations can also occur in its subnetwork. Connection restoration may produce conflicting concurrent model updates. We rely on vector clocks to detect these conflicts and on the conflicting model updates themselves. Experiment #3 aims at checking the behaviour of our algorithm in this conflicting updates situation.

Experiment protocol The experiment protocol is based on experiment #2. We use a similar grid architecture but with only 12 nodes. An initial reconfiguration (c1)

is launched on the *p00* node just after the bootstrap phase. All network links are up. Then a fault is simulated on the link between nodes *p00* and *o00*. Nodes *o00*, *o01*, *o02* are then isolated. A new model is then pushed on node *p00* (c2) and a different one on node *o00* (c3). A delay of 1000 ms separates each reconfiguration and the algorithm is configured with a notification and a pooling period of 2000 ms.

Analysis Figure 8 shows results of experiment #3, which are derived from our branching algorithm traces. Three reconfigurations are represented as a succession of segments that show the propagation of updates. The first reconfiguration on the healthy network is represented in black (at time 0). Reconfiguration pushed on *o00* (at time 2500) is represented in blue and the second reconfiguration pushed on *p00* (time 2000) in red. The first reconfiguration propagates seamlessly to all nodes. At time 1500 a network failure is simulated. The second model given to *p00* is propagated to all nodes except nodes reachable through *o00* only. Similarly, the second model pushed on node *o00* is not propagated to nodes after *p00*. At time 8000 we cancel the network failure simulated at time 1500. After a synchronization delay (380ms) we observe the branching of the two concurrent models as well as propagation of the merged version (purple line).

5 Discussion and related work

Our approach is dedicated to model at runtime synchronization, and combines commonly used paradigms in distributed computing like vector clocks (e.g. used in distributed hash table frameworks) and gossiping (e.g. used in social network graph dissemination). This section discusses our experimental results and compare them to other related work.

Vector clock size. Our first experiment measures the size of data exchanged during the reconfiguration step, as well as the time required to perform this reconfiguration. Figure 5 shows that the model@runtime synchronization overhead is significant, and this is mostly due to vector clock size. Many studies aim at reducing the data size of vector clocks, especially when synchronizing an unbounded number of peers. Sergio and al [1] proposed the Interval Tree Clocks to optimize the mapping between the node identifier and its version. Our algorithm takes advantage of the model payload to allocate dynamic identifiers to nodes. Data such as node names or network identifications are stored in the payload itself and with this information we can already improve vector clocks. However, we plan to implement the interval tree clocks' fork and join model in the future. The size of exchanged data depends on the number of nodes and therefore modularization techniques are needed to maintain scalability and manage large mesh networks. Our approach addresses this need by exploiting the group structure of Kevoree. Each group instance synchronizes with a subset of nodes only, to keep the size of the vector clock under control.

Distributed reconfiguration capability. Concurrency management is a key problem in distributed systems. Many peer to peer systems solve it by having a single point of update for a given piece of data, limiting concurrent access to a one writer/many readers situation for that data. Realistic distributed configuration management is a many writers/many readers situation, because reconfigurations often involve more than one node. The simplest solution to this problem would use a single point for new configuration computation and dissemination start. As it avoids concurrency, such a system has a central point

of failure incompatible with our use case. More advanced approaches such as the one proposed in [17] use distributed coordination techniques such as consensus to build the new configuration. They proposed an approach that allows the distributed nodes to collaborate to build the new configuration. Each node is responsible for building its local configuration. Configuration propagation is then done using a gossip algorithm without the need of concurrency management, since new configurations can be disseminated from a single originator node only. This approach based on a single source is unusable in our use case, because the sporadic nature of the nodes prevent their participation in a global consensus. On the contrary, our technique presented in this paper lets the distributed configuration evolve freely, even for nodes are isolated in unreachable groups. Every node can then compute a new global model that can be issued concurrently. Some approaches in distributed hash table implementations also rely on fully distributed data dissemination, e.g. Voldemort, where table modifications can occur in several nodes. This allows for service operation in degraded mode in the case of node disconnections. However, concurrency management must be managed separately. GossipKit [12] proposes a generic framework to evaluate and simulate gossip-derived algorithms. The project contains a minimal extensible event-based runtime with a set of components to define dedicated gossip protocols. We plan to integrate the GossipKit API in order to evaluate our algorithm on a GossipKit simulator.

Inverted communication and propagation delay. In our approach we reverse the traditional communication strategy of a gossip algorithm (*push* approach). New configurations are not directly pushed to the neighbours but they are stored instead, waiting for an active synchronisation by the neighbour (*pull* approach). This strategy lessens the impact of down network links on propagation delay, as shown by our experiment results on Figure 7. In addition, this enables message replay because a configuration is stored until neighbor connectivity is reestablished. These two properties are particularly useful for unreliable mesh networks. However, pull approaches have higher propagation time, but when combined with an observer pattern (a lazy push/pull approach) our results show that the gains are significant while keeping the interesting properties of pull. This experimental result is consistent with Leitao *et al* [11], which details several communication strategy for gossip algorithms.

6 Conclusion

In this paper we proposed a peer to peer and distributed dissemination algorithm to manage dynamic architectures based on the *models at runtime* paradigm. This algorithm is part of a larger framework that manages the continuous adaptation of pervasive systems. Using experimental results we have shown how our approach enhances reliability and guarantee of information delivery, by mixing and specializing different distributed algorithms. Our propagation algorithm relies on its payload (a model of the system) to overcome limits of vector clocks and to handle peer to peer concurrency conflicts. Thanks to the protocol layer based on vector clocks, a system architecture model propagated by the algorithm is always consistent, even on complex mesh network topologies. When concurrent updates are detected, the model at runtime layer is able to reconcile these updates to provide a valid architecture. By allowing each node to compute a new

configuration, our approach supports dynamic adaptation on peer to peer networks without any central point of failure. This experimental demonstration of resilience on sporadic networks allows integration of our approach into adaptive architectures such as a firefighters tactical information system. In this direction, we are currently designing a dynamically scalable tactical information system in collaboration with a department of firefighters of Brittany; this system is a multi-user, real time decision system for incident management.[4].

Acknowledgment The research leading to these results has received funding from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement 215483 (S-Cube).

References

1. P.S. Almeida, C. Baquero, and V. Fonte. Interval tree clocks: A logical clock for dynamic systems. *Principles of Distributed Systems*, page 259274.
2. R. Baldoni, M. Raynal, and U..R.L.S. DIS. Fundamentals of distributed computing. *IEEE Distributed Systems Online*, 3(2):1–18, 2002.
3. Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@runtime. *IEEE Computer*, 42(10):22–27, 2009.
4. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J-B. Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.
5. C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Luna-Aceves. A loop-free extended bellman-ford routing protocol without bouncing effect. *SIGCOMM Comput. Commun. Rev.*, 19:224–236, August 1989.
6. P.T. Eugster, R. Guerraoui, A.M. Kermarrec, and L. Massoulié. From epidemics to distributed computing. *IEEE computer*, 37(5):60–67, 2004.
7. C.J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th ACSC*, volume 10, pages 56–66, 1988.
8. Ralph Johnson and Bobby Woolf. The Type Object Pattern, 1997.
9. Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
10. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.
11. J. Leitão, J. Pereira, and L. Rodrigues. Gossip-based broadcast. *Handbook of Peer-to-Peer Networking*, pages 831–860, 2010.
12. S. Lin, F. Taïani, and G. S. Blair. Facilitating gossip programming with the gossipkit framework. In *DAIS*, 2008.
13. F. Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, pages 215–226, 1989.
14. B. Morin, O. Barais, J-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.
15. G.S. Raj, PG Binod, K. Babo, and R. Palkovic. Implementing service-oriented architecture (soa) with the java ee 5 sdk. *Sun Microsystems, revision*, 3, 2006.
16. R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102. IEEE, 2001.
17. D. Sykes, J. Magee, and J. Kramer. Flashmob: distributed adaptive self-assembly. In *Proceeding of the 6th SEAMS*, pages 100–109. ACM, 2011.

[4] More details on this project can be found in `http://kevoree.org/related_projects`