Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems

Xiaoxing Ma*†, Luciano Baresi*, Carlo Ghezzi*, Valerio Panzica La Manna*, Jian Lu†
xxm|lj@nju.edu.cn, baresi|ghezzi|panzica@elet.polimi.it

*Politecnico di Milano, Dipartimento di Elettronica e Informazione, 20133 Milano, Italy
†Nanjing University, State Key Laboratory for Novel Software Technology, 210093 Nanjing, China

ABSTRACT

There is an increasing demand for the runtime reconfiguration of distributed systems in response to changing environments and evolving requirements. Reconfiguration must be done in a safe and low-disruptive way. In this paper, we propose version consistency of distributed transactions as a safe criterion for dynamic reconfiguration. Version consistency ensures that distributed transactions be served as if there were operating on a single coherent version of the system despite possible reconfigurations that may happen meanwhile. The paper also proposes a distributed algorithm to maintain dynamic dependences between components at architectural level and enable low-disruptive version-consistent dynamic reconfigurations. An initial assessment through simulation shows the benefits of the proposed approach with respect to timeliness and low degree of disruption.

Categories and Subject Descriptors

D.2.7 [**Distribution, Maintenance, and Enhancement**]

General Terms

design, reliability, management

Keywords

Dynamic reconfiguration, Version-consistency, Component-based distributed system

1. INTRODUCTION

Oftentimes component-based distributed systems (CBDSs) must cope with changes in the environment in which they are embedded and in the requirements they must satisfy. Such changes may be hard to predict at design time or they may be too expensive to identify and handle by the initially designed software. It is therefore necessary to modify existing parts of the implementation or add new functionality at runtime without blocking the system.

ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.

Compared to off-line maintenance, dynamic modifications are more difficult since in addition to the correctness of the new version, they must also preserve the correct completion of on-going activities. At the same time, they should also minimize the interruption of system's service (usually called *disruption*) and the delay with which the system is updated (also called *timeliness*).

Dynamic reconfigurations (of CBDSs) are not trivial [15, 25, 27]. A safe approach that avoids the direct manipulation of application-specific states enables the reconfiguration of components after they have reached a *quiescence* status [15]. This approach has the advantage of a clear separation of concerns between computation and architectural (re)configuration. But since reconfiguration is oblivious to application states, it must be conservative. It blocks all potentially dependent computations before enabling any reconfiguration to ensure consistency, and thus it may bring more disruption than necessary.

The quiescence-based approach only considers *static* dependences between components specified by architectural configurations. Since these dependences pessimistically include all *potential* constraints among transactions, one can reduce disruption by considering *dynamic* dependences. These are temporal relationships between components caused by on-going transactions, and they only indicate the *current* constraints on the reconfigurability of the system. Multiple transactions are allowed to run during reconfiguration, thus reducing the degree of disruption and improving the timeliness of the reconfiguration.

Existing proposals ([4, 7, 25]) only use dynamic dependences to ensure some local consistency properties. In contrast, the approach presented in this paper exploits them to ensure the "global" consistency of multi-party distributed transactions through the notion of *version consistency.* Dynamic reconfigurations are seen as (sequences of) runtime updates of components; version consistency ensures that every transaction be entirely served by either the old versions of system's components or by the new ones, no matter when the reconfiguration happens. This approach introduces less disruption than the quiescence-based one since a component can be updated even when it is not quiescent, but it has not been used yet or it will not be used anymore by any on-going transaction.

Besides *version consistency* as a new criterion for the safe dynamic reconfiguration of CBDSs, the paper also proposes a management framework for multi-party distributed transactions. The framework exploits dynamic dependencies, a distributed algorithm to update the dependency model ac-

cordingly at runtime, and a locally checkable condition that is sufficient to ensure version-consistent dynamic reconfigurations.

The rest of the paper is organized as follows. Section 2 introduces our model of CBDS and the challenges posed by dynamic reconfiguration. Section 3 highlights the limitations of the two most representative approaches and paves the ground for our proposal. Section 4 presents version consistency and the management framework. Section 5 summarizes the main results of the assessment conducted to evaluate the proposal. Section 6 surveys related approaches and Section 7 concludes the paper.

2. PROBLEM SETTING

Similarly to UML component diagrams, a component-based distributed system may be described as a set of components (*nodes*) with in-ports and out-ports. *Components* are linked by directed edges from out-ports to in-ports. The resulting directed graph is called the system's static *configuration*, where each node is tagged with the current version of the component it represents. A static configuration can be described as follows:

Definition 1 (Static Configuration). *A static configuration of a component-based distributed system is a directed graph whose nodes represent versioned components with in-ports and out-ports. A directed edge, from the out-port of a given node N to the in-port of other node M, represents a static dependence, that is, the possibility for N to require a service provided by M.*

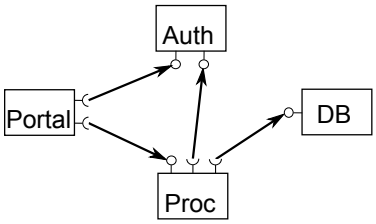

Figure 1: Our example system.

Figure 1 shows an example system used throughout the paper. A portal component (Portal) interacts with an authentication component (Auth) and a business processing component (Proc), while Proc interacts with both Auth and a database component (DB). This means that Portal statically depends on Proc and Auth, and Proc depends on Auth and DB.

Our approach supports the dynamic reconfigurations of CBDSs with transactional behavior, that is, systems whose execution evolves through atomic, separate tasks (transactions). This model is largely adopted in many applications: from systems implemented on top of component-based infrastructures (like EJBs or Spring) to service-oriented applications.

A *transaction* is a sequence of actions executed by a component that completes in bounded time. Actions include local computations and message exchanges. Transactions are described by the rectangles $T_0..T_4$ in the sequence diagram shown in Figure 2, which is used hereafter as an example. A transaction T can be initiated by an outside client or by another transaction T'. T is called a *root transaction* in the former case and a *sub-transaction* (of T') in the latter case. If a transaction T initiates sub-transactions, T, along with its sub-transactions, is said to be a *distributed transaction.* For example, T_4 is a sub-transaction of T_2 in the sequence diagram of Figure 2. The host component (node) of transaction T is denoted as h_T.

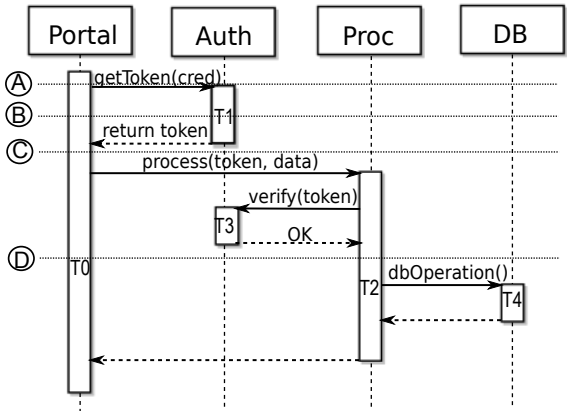

Figure 2: Detailed scenario.

The term $sub(T_1, T_2)$ denotes that T_2 is a direct sub-transaction of T_1. A transaction can only be a direct sub-transaction of one transaction. The set $ext(T) = \{x | x = T \vee sub^+(T, x)\}$ is the *extended transaction set* of T, which contains T and all its direct and indirect sub-transactions. The extended transaction set of a root transaction models the concept of *distributed transaction* that can span over multiple components. By definition, a root transaction is not a sub-transaction of any other transaction. A root transaction has a unique, system-wide identifier. For each transaction T, h_T must know $root(T)$, which is the identifier of the root transaction of T.

A transaction can initiate a sub-transaction on a neighbor component only when the host component of the initiating transaction is statically dependent on the neighbor component in charge of executing the sub-transaction. For example, the sequence diagram of Figure 2 illustrates a distributed transaction, where root transaction T_0 initiates T_1 and T_2. In turn transaction T_2 initiates T_3 and T_4. We also assume that transactions are also always notified of the completion of their sub-transactions and that a transaction cannot end before the termination of its sub-transactions. The other messages exchanged between h_T and h_{T_i} —because of T and its direct sub-transaction T_i— are temporally scoped between the two corresponding messages that initiate T_i and notify its completion.

Figure 2 shows a detailed usage scenario for the example system. The Portal first gets an authentication token from Auth and then uses it to require the service from Proc. Proc verifies the token through Auth and then starts computing, and interacting with DB. If we consider the root transaction T_0 at Portal, its extended transaction set is $ext(T_0) = \{T_0, T_1, T_2, T_3, T_4\}$, where T_1 at Auth is in response to the getToken request, T_2 at Proc in response to process, T_3 at Auth in response to verify, and T_4 at DB for T_2's request of database operations.

Different transactions may run concurrently in the system.

If the executions of two transactions on the same component do not interfere with each other, we assume they are safe. In contrast, if they compete for some resources, we rely on the *isolation* [11] provided by components to regulate the access.

2.1 Runtime updates

A dynamic reconfiguration policy defines *when* the update can be performed and *how* it must be accomplished to keep the system consistent.

A dynamic reconfiguration is an atomic sequence of runtime updates of components[1]. More rigorously, an update is specified as a tuple $< \Sigma, \omega, \omega', \mathcal{T}, s >$, where Σ is the original system's configuration and ω is the set of components that must be substituted by the new versions in ω'. The update happens when the system is in state s, and the state transformer $\mathcal{T}$ trasforms s into $s' = \mathcal{T}(s)$, which is a state of the updated system with configuration $\Sigma' = \Sigma[\omega'/\omega]$[2]. The system is expected to continue from s' with no externally-visible erroneous behavior. The (global) state of a CBDS comprises the local states of all components and all messages in transit. The state transformer is defined on global states, but in practice one can hardly manipulate states other than those local to the to-be-updated components.

We assume that given a runtime update $< \Sigma, \omega, \omega', \mathcal{T}, s >$, the corresponding off-line update $< \Sigma, \omega, \omega' >$ is correct. This means that the transactions running on Σ satisfy the old system specification $\mathbb{S}$ and the transactions on $\Sigma' = \Sigma[\omega'/\omega]$ satisfy the new specification $\mathbb{S}'$. Given the distributed nature of these transactions we can only adopt a *weak* definition of *correctness of a runtime update*, which is defined as follows:

- The transactions that end before the update satisfy $\mathbb{S}$;
- The transactions that begin after the update satisfy $\mathbb{S}'$;
- The transactions that begin before the update, and end after it, satisfy either $\mathbb{S}$ or $\mathbb{S}'$.

For example, if we consider the running example, one may assume that Auth be updated to exploit a stronger encryption algorithm to prevent security threats. Although the new algorithm is incompatible with the old one, the other components need not to be updated because all encryption/decryption operations are done within Auth. The specification of all other transactions remains unchanged; the off-line update would be "easy" and we assume it to be correct. The problem is that if we update Auth at runtime, we should ensure that all running transactions execute correctly before and after the update. If the update were allowed to happen any time, it would be difficult to ensure it [12]. An obvious restriction on *when* the update can happen is to impose that components targeted for update be *idle*.

Definition 2 (Idle Component). *A component is idle iff it is not currently hosting transactions and its current local state is equivalent to the initial one.*

[1]We consider any architectural change that can be treated as a replacement of components. For example, service rebindings can be seen as updates of the components that implement the services.

[2]This is to say that Σ' is equal to Σ where all components in ω have been substituted by the new versions in ω'.

Note that if we restrict updates to occur when components are idle then $\mathcal{T}$ only needs to be locally defined on the initial state of all substituted components. The initial state of the new component is assumed to be "equivalent" to the current state of the idle component being replaced.

The assumption that components can only be updated when they are idle is often insufficient to guarantee safe runtime updates. In fact, if we consider the scenario of Figure 2, and substitute Auth when idle, but after serving getToken (after time ©), the resulting system would behave incorrectly since the security token would be created with an algorithm and validated by another.

Since the correctness of arbitrary runtime updates is undecidable (even if the corresponding off-line update is correct and the runtime update only happens when to-be-updated components are idle [17]), we can only derive some automatically checkable conditions that are sufficient for the correctness by scoping the class of updates we consider. These conditions must be: (a) strong enough to ensure the correctness of runtime updates, (b) weak enough to allow for low-disruptive and timely changes, and (c) automatically checkable in a distributed setting (without enforcing unnatural centralized solutions).

3. QUIESCENCE AND TRANQUILLITY

In a seminal paper, Kramer and Magee [15] proposed a criterion called *quiescence* as a sufficient condition for a node to be safely manipulated in dynamic reconfigurations. They also model a distributed system as a directed graph, and define a transaction as "an exchange of information between two and only two nodes, initiated by one of the nodes. Transactions are the means by which the state of a node is affected by other connected nodes in the system. Transactions consist of a sequence of one or more message exchanges between the two connected nodes. It is assumed that transactions complete in bounded time and that the initiator of a transaction is aware of its completion". A transaction T may also depend on other (consequent) transactions T_i: the completion of T depends on the completion of all the T_i. This definition of transaction corresponds to our definition of distributed transaction between only two nodes. Through the notion of dependency between transactions, however, they also capture the concept of a distributed transaction involving multiple nodes.

Definition 3 (Quiescence). *A node is quiescent if:*

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.*

A node satisfying the first two conditions is said to be *passive*. A node is required to respond to a passivate command from the configuration manager by driving itself into a passive state in bounded time. The last two conditions further make the node independent of all existing or future transactions, and thus it can be manipulated safely. To drive a node into a quiescent status, in addition to passivating it, all the nodes that statically depend on it must also be passivated to ensure the last two conditions.

According to this approach, a node cannot be quiescent before the completion of all the transactions initiated by statically dependent nodes. This means that the actual update could be deferred significantly. In our example, Auth cannot be quiescent before the end of the transactions hosted by Portal and Proc (T_0 and T_2). Moreover, all the other nodes that could potentially initiate transactions, which require service from Auth, directly or indirectly, are passivated, and their progress blocked till the end of the update. Again, in our example Portal and Proc are to be passivated before changing Auth. This means that the adoption of this approach could introduce significant disruption in the service provided by the system.

To reduce disruption, Vandewoude et al. [25] proposed the concept of *tranquillity*, as alternative to quiescence. The idea is that there is no need for waiting a transaction to complete if it will not request the service provided by the node targeted for update, even if the node has been involved in the transaction. It is also permitted to update a node even if some on-going transactions will require the service provided by the node in the future, but they have not interacted with it yet.

Definition 4 (TRANQUILLITY). *A node is tranquil if:*

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;*
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.*

While claimed to be "a sufficient condition for application consistency during a dynamic reconfiguration" [25], the notion of tranquillity is based on a rather strong assumption. If we used the definition of tranquillity in our model, the notion of distributed transaction could not be fully expressed as defined in Section 2. In fact, under the assumption imposed by tranquility, a distributed transaction initiated by a root transaction T at node N, could only contain subtransactions hosted by (adjacent) nodes directly connected to N. This means that a sub-sub-transaction (i.e., a subtransaction initiated by another sub-transaction), hosted by a node that is not directly connected to N, would not be part of the distributed transaction, and thus it could use any version of the components since it is an independent entity.

This limitation would permit unsafe updates in the scenario of Figure 2. In fact, after Auth returns the token to Portal, it will not participate in the session initiated by Portal anymore. In addition, before the request for verification is sent, Auth has not participated in the session initiated by Proc, and so Auth is tranquil at time ©. However, if Auth were updated at this time, the verification would fail because the token was issued by the old version of Auth, which might use a different encryption algorithm to validate the security token. This failure would not happen if the system entirely complied with either the old or the new system configuration.

To conclude, we can say that the quiescence-based approach is a general and safe solution, but it can be highly disruptive. The tranquillity-based approach is less disruptive, but its assumption is too restrictive to be applicable to a wide set of systems (e.g., our example). Our goal is to get the best of the two proposals and add efficiency and timeliness to safety.

4. DYNAMIC RECONFIGURATION

The notion of version consistency is proposed as a sufficient condition for the safety of dynamic reconfigurations.

Definition 5 (VERSION CONSISTENCY). *Transaction T is version consistent with respect to an update $< \Sigma, \omega, \omega', \mathcal{T}, s >$ iff $\nexists T_1, T_2 \in ext(T) \mid h_{T_1} \in \omega \wedge h_{T_2} \in \omega'$. A dynamic reconfiguration caused by an update $< \Sigma, \omega, \omega', \mathcal{T}, s >$ is version consistent if all transactions hosted by the current configuration Σ are version consistent.*

The definition is justified by the fact that we assume the new configuration to be correct and by the fact that any extant transaction with all its (direct and indirect) subtransactions is entirely executed in the old or in the new configuration. Also note that a transaction that ends before (starts after) the update cannot have a direct or indirect subtransaction hosted by the new (old) version of a component being updated.

For our example, if the update of Auth happens after transaction T_0 begins but before it sends a getToken request to Auth (time Ⓐ), all transactions in $ext(T_0)$ (i.e., all transactions in Figure 2) are served in the same way as if the update happened before they all began. If it happens at any time after Auth replies to the verify request issued by Proc (time Ⓓ), all transactions in $ext(T_0)$ are served the same way as if the update happened after they all ended. However, if it happens at time Ⓒ, then h_{T_1} = Auth, but h_{T_3} = Auth′. As both T_1 and $T_3 \in ext(T_0)$, T_0 would not be version-consistent.

In general, the tranquillity-based approach is not sufficient for version-consistent dynamic reconfigurations. It is equivalent to limiting the extended transaction set to $ext_{tran}(T) = \{x | x = T \vee sub(T, x)\}$. It only ensures a kind of local consistency, but not the global consistency of entire distributed transactions. The quiescence-based mechanism —if one takes dependent transactions into account— ensures version consistency because no distributed transactions depending on the node to be updated can exist when the node is quiescent. However, it would take a too pessimistic attitude.

4.1 Dynamic dependences

Since version consistency is not directly checkable, we need to identify a condition that is checkable on a component (or a set of components) and ensure that its (their) runtime update does not break version consistency; this condition must also allow for low-disruptive and timely reconfigurations.

Dynamic dependences are the means to define such a condition, and they can easily be added to the diagram of Figure 1 through properly-labelled edges. Dynamic edges, represented as dashed arcs, are added and removed dynamically and they are labelled as either future or past. A future edge represents the possibility for the source node to initiate a transaction on the target node; a past edge witnesses the fact that a transaction initiated by the source node has already been executed on the target node. Future and past edges are also labelled with the identifier of a root transaction. We use $C \xrightarrow[T]{future(past)} C'$ to denote a future (past)

edge labelled with the identifier of root transaction T, from component C to component C'.

Definition 6 (Valid Configuration). *A static configuration decorated with edges representing dynamic dependences between corresponding components, hereafter, a configuration, is valid iff future and past edges are created and removed at runtime according to the following constraints:*

1. (host-validity) *The hosting component C of a transaction T is augmented with a pair of edges $C \xrightarrow[root(T)]{future} C$ and $C \xrightarrow[root(T)]{past} C$ as soon as T is initiated and till it ends;*
2. (locality) *Any future edge $C \xrightarrow[T]{future} C'$ (or past edge $C \xrightarrow[T]{past} C'$) cannot exist without a static edge $C \rightarrow C'$ connecting the same two nodes;*
3. (future-validity) *A future edge $C \xrightarrow[T]{future} C'$ must be added before the first sub-transaction $T' \in ext(T)$, with $T' \neq T$, is initiated, and it cannot be removed as long as transactions hosted by C will initiate further $T'' \in ext(T)$ on C'.*
4. (past-validity) *A past edge $C \xrightarrow[T]{past} C'$ must be added at the end of a transaction $T' \in ext(T)$ on C', initiated by a transaction hosted on a statically dependent node C and cannot be removed at least until the end of T.*

Figure 3 shows some configurations of the example system (two concentric cycles correspond to a pair of local future/past edges labelled with T_0). The configuration of Figure 3(a) corresponds to time Ⓐ in Figure 2: transaction T_0 just began on Portal. The dynamic edges indicate that (i) there may be a transaction in $ext(T_0)$ running on Portal; (ii) in order to serve transactions in $ext(T_0)$, Portal might use Auth and Proc in the future, and also Proc might use Auth and DB. Figure 3(b) corresponds to time Ⓑ and says that a transaction in $ext(T_0)$ (T_1) is currently running on Auth, but no further transactions in $ext(T_0)$ hosted by Portal will initiate any sub-transaction on Auth anymore because there is no T_0-labelled future edge between the two nodes. Figure 3(c), which corresponds to time Ⓒ in Figure 2, indicates that Auth might have hosted transactions in $ext(T_0)$ initiated by Portal in the past, and might host other transactions in $ext(T_0)$ initiated by Proc in the future. Figure 3(d) corresponds to time Ⓓ in Figure 2 and shows that Auth, although it might have hosted transactions in $ext(T_0)$, is not hosting and will not host them anymore.

Note that to keep a configuration valid, and consider alternative execution flows, one can adopt a conservative management of future/past edges between components. For example, if T_2 on Proc needed to evaluate the results from DB before deciding whether to ask Auth for verification, one could keep the future edge from Proc to Auth till T_2 does not need Auth anymore. In the worst case, this could be till the end of T_2.

On the other hand, with the validity of the configuration preserved, future (past) edges should be removed (created) as early (late) as possible to reduce spurious dependencies between components that could hinder actually legitimate dynamic reconfigurations. For instance, the future edge from Portal to Auth in Figure 3(a) is removed once Portal

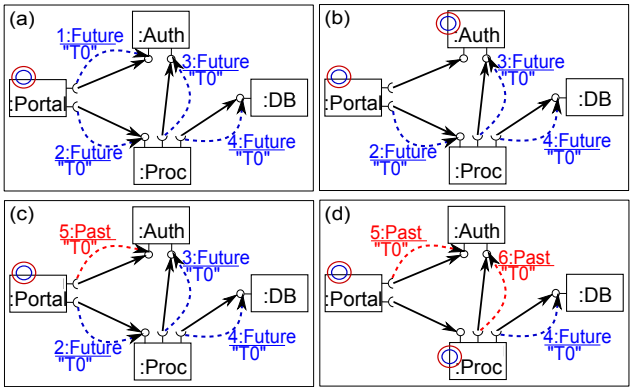

Figure 3: Some configurations of the example system with explicit dynamic dependences.

realizes that T_0 will not initiate sub-transactions on Auth anymore — this happens just after T_0 initiates T_1.

Given a valid configuration, we can identify a locally checkable condition that is sufficient for the version consistency of dynamic reconfigurations.

Definition 7 (Freeness). *Given a configuration Σ, a component C (or a set of components ω) is said to be free of dependences with respect to a root transaction T iff there does not exist a pair of T-labelled future/past edges entering C (or ω). C (or ω) is said to be free in Σ iff it is free with respect to all the transactions in the configuration.*

Auth is free of dependences with respect to T_0 in the configurations of Figure 3(a) and 3(d), while the local dynamic edges at Auth trivially falsify its freeness in the configuration of Figure 3(b). Intuitively, for a valid configuration Σ, the freeness of a component C with respect to a root transaction T means that the distributed transaction modeled by $ext(T)$ either has not used C yet (otherwise there should be a past edge), or will not use C anymore (otherwise there should be a future edge). This leads to the following proposition.

Proposition 1. *Given a valid configuration Σ of a system, a dynamic reconfiguration of a set of its components ω is version consistent if it happens when ω is free in Σ.*

Proof. If we suppose that version consistency does not hold, there is a transaction T that is not version-consistent, that is, $\exists\, T_1, T_2 \in ext(T) \mid h_{T_1} \in \omega \ \wedge \ h_{T_2} \in \ \omega'$.

1. There is no ongoing transaction hosted by any $C \in \omega$ when the update happens. This follows from the host-validity of Σ and the freeness of components in ω.
2. Let $\bar{\omega}$ be the set of unchanged components. Because T begins no later than T_1 and T_1 begins before the update since $h_{T_1} \in \omega$, T begins before the update. Similarly T ends after the update because of T_2. Since no transaction is hosted by any components in ω when the update happens, h_T must be in $\bar{\omega}$.
   (a) Consider the sub-transaction chain from T to T_1. There must be such T_i and T_j that $sub(T_i, T_j) \wedge h_{T_i} \in \bar{\omega} \wedge h_{T_j} \in \omega$. According to the past-validity of Σ and the fact that T_j must have already ended when the update happens (see point 1), there is a past T-labelled edge entering h_{T_j}.

(b) Because of the validity of the static configuration after the update, ω' must inherit all the incoming static dependence edges from ω. Without any loss of generality, let T_2 be the first transaction in $ext(T)$ initiated on a component in ω'. Consider the sub-transaction chain from T to T_2. There must be a T_k such that $sub(T_k, T_2) \wedge h_{T_k} \in \bar{\omega}$. Before T_k initiates T_2, all transactions from T to T_k in the chain will not be affected by the update because they are independent of transactions not in $ext(T)$, and T_2 is the first transaction in $ext(T)$ hosted by a component in ω'. According to the future-validity of Σ and the fact that T_2 is still to be initiated when the update happens, there is a future T-labelled edge from h_{T_k} to a component in ω.

So there is a pair of T-labelled past/future edges entering ω. This contradicts the freeness of ω. □

4.2 Distributed management of dependences

An advantage of specifying dynamic dependences with future and past edges is that the validity of a configuration can be achieved through the cooperation of components. Each component makes its decisions locally with limited information about the application logic; checking for freeness is also local to the part of the system to be updated.

The definition of *valid configuration* gives lower bounds for the time intervals during which dynamic dependence edges should exist to keep a configuration valid. One can satisfy all the conditions straightforwardly by creating all the dynamic edges at the beginning of a root transaction and by removing them at the end of it. However, although version consistency would be ensured, disruption could be too high.

We assume that given a transaction T, its host component h_T knows $f(T)$, the set of static edges through which it might initiate sub-transactions on neighbor components in the future, and $p(T)$, the set of static edges through which it has initiated sub-transactions in the past[3].

The overall configuration with dynamic dependences is maintained in a distributed way. Each component only has a local view of the configuration that includes itself and its direct neighbors. A component is responsible for the creation and removal of the dynamic edges leaving from it, but it is also always notified of the creation and removal of the dynamic edges entering it. Components exchange management messages, besides those related to transactions, to keep the consistency among the views of neighbor components. The maintenance of the distributed configuration may slightly delay the execution of the actual transactions, but it guarantees that no transactions will be blocked forever.

4.2.1 Management algorithm

The management of dynamic edges for different distributed transactions is independent of each other since these edges are labelled with the identifiers of the corresponding root transactions. If we consider a distributed transaction $ext(T)$, our algorithm consists of three steps: **set up**, **progress**, and **clean up**. Through these steps, the locality of the configuration is ensured by only creating dynamic edges that pair the existing static ones, and the host validity is preserved by always creating local future and past edges (if they do not already exist) when a transaction is initiated and by removing them only after the end of the transaction (if there are no other on-going transactions that need these edges).

In more detail, the three steps carry out the following actions and help preserve future and past validity as follows:

Set up: This step is carried out when the root transaction T is initiated and before it initiates any sub-transaction. During this phase, h_T creates a future edge for each of its out-going static edges that T might use to initiate sub-transactions according to $f(T)$, notifies corresponding neighbor components, and waits for their acknowledgements. Only after receiving all these acknowledgements, T is allowed to initiate its sub-transactions.

As soon as a component C is notified of the creation of an incoming future edge $fe = C' \xrightarrow[T]{future} C$ by C', it starts creating its own future edges, notifies neighbor components, waits for their acknowledgements, and then acknowledges C' back. The rationale is that by accepting the creation of fe, C "promises" C' to host some $T_C \in ext(T)$ in the future, but to make such a promise C first needs to get the promises from those components that T_C might need to use. Note that loops in the static configuration can be handled by avoiding the creation and notification of duplicated edges. This conservative way creates future edges to achieve a valid configuration with respect to T when the set up step finishes. Figure 3(a) shows the result of setting up future edges for transaction T_0 of Figure 2.

Progress: Due to the execution of transactions in $ext(T)$, future edges are gradually removed as soon as the algorithm knows that a component *will-not-use* another one anymore, and past edges are created to register *have-used* relationships.

In more detail, new *will-not-use* information can be available at various time instants including (1) when a transaction in $ext(T)$ hosted by C successfully initiates a sub-transaction on a neighbor component; (2) when a transaction in $ext(T)$ hosted by C ends; and (3) when C is notified of the removal of a T-labelled incoming future edge. No matter when the information is acquired, a non-local future edge $fe = C \xrightarrow[T]{future} C'$ is removed only when: (1) no on-going $T' \in ext(T)$ hosted by C will initiate any sub-transaction on C' through $C \xrightarrow{static} C'$ anymore; and (2) there is no incoming T-labelled future edge entering C, that is, no further $T'' \in ext(T)$ will be initiated on C anymore. This condition ensures future validity because once fe is removed, no sub-transactions in $ext(T)$ need to be initiated through $C \xrightarrow{static} C'$ anymore.

To record the *have-used* information, when a sub-transaction T originally initiated by T' ends, a corresponding past edge $pe = h_{T'} \xrightarrow[root(T)]{past} h_T$ is created *immediately*. This is done by letting h_T first notify $h_{T'}$ the end of T, and by removing the corresponding local edges only when pe is created by $h_{T'}$. This hand-shaking ensures that past validity holds despite the asynchrony of messages.

[3]It is safe to over estimate $f(T)$ and $p(T)$, but better accuracy means better timeliness and less disruption in the dynamic reconfiguration. This information can be extracted automatically from the component, but also by monitoring its transactions; the degree of accuracy of this information is up to the user. Note that similar information is also required to decide the actual tranquillity of components [25].

In our example scenario, Portal removes the future edge to Auth after it initiates T_1 on Auth as it is sure that T_0 will not initiate such transactions anymore (Figure 3(b)). When T_1 ends, Portal immediately creates a past edge to record the fact that it has used Auth (Figure 3(c)). Eventually, the system reaches the configuration of Figure 3(d), where Auth is free with respect to T_0.

Clean up: This step is carried out only when T ends. The algorithm recursively removes all remaining past and future edges. This step does not affect the validity of the configuration.

For the sake of readability, the algorithm has been described by taking the viewpoint of a distributed transaction, but the actual algorithm runs on each component without any centralized control. The underlying message delivery is assumed to be reliable, and the messages between two components are kept in order. Interested readers can refer to [17] for all details. Also note that since tracking dynamic dependences across the whole system can be expensive and unnecessary, a user can specify a portion of the system as scope for version consistency if the impact of the update is known to be limited. More details on this issue are also discussed in [17].

4.2.2 On-demand set up

The above algorithm assumes that configurations are always kept up-to-date no matter whether there is a request for dynamic reconfiguration. However, configuration management could introduce significant overhead. If reconfigurations are rare, a valid configuration can be set up on demand only when a request is planned or expected. To this end, dynamic edges must be created both for new transactions and for on-going ones.

The set up works as follows. The working *mode* of each component can be: NORMAL, ONDEMAND, or VALID. NORMAL means that the component is not supposed to manage dynamic dependences, while VALID requires it. ONDEMAND imposes that the component: (1) manages the dynamic dependences for all the new root transactions initiated locally, (2) blocks the initiation and termination of locally hosted sub-transactions temporarily, and (3) for each locally-hosted ongoing root transaction T, creates future (past) edges towards the components that might host in the future (might have hosted in the past) transactions in $ext(T)$ in a way similar to the one described above.

At the beginning, all components operate in mode NORMAL. When a component targeted for update receives a request for reconfiguration, it sends a request for set up to itself and waits for its mode to become VALID before using the approach we describe next to achieve freeness. Upon receiving a request for set up, a component C switches to mode ONDEMAND and sends set up requests to all the components C_i that statically depends on it. Once C has finished the set up of dynamic edges for its local root transactions, and has received all the acknowledgements from C_i, it switches to mode VALID, resumes all blocked transactions, and sends confirmations to all the nodes from which it received requests for set up. The whole on-demand set up procedure completes when the targeted component becomes VALID.

On-demand set up of dynamic dependencies also makes dealing with dynamic reconfigurations resulting in new dependencies straightforward, and thus multiple reconfiguration steps in sequence are supported. However, although our approach allows multiple components to be updated together (in a single reconfiguration step), it does not currently support concurrent reconfigurations, meaning independent updates of different components happening at same time without coordination.

4.3 Achieving freeness

Given a valid configuration maintained at runtime, checking for the freeness of ω —the set of components targeted for update— is straightforward since the condition is local to ω, but freeness can be achieved in different ways.

The first strategy, called *waiting for freeness (WF)*, is simply opportunistic. The system just waits for freeness to manifest itself. This strategy has no extra overhead other than setting up and maintaining the valid configuration and checking for the freeness of ω. However, although every transaction completes in finite time, the freeness of ω could never be reached —for example, there could always be transactions running on some components in ω.

To ensure the timeliness of runtime updates, a second strategy, called *concurrent versions (CV)*, lets the components in ω and ω' co-exist during the update process. The use of multiple versions of the same components is not new (e.g., HERCULES [8] and Upstart [1] adopt it), but our proposal can add version consistency as means to choose which version must serve a request and to decide when a version can be deleted. Given a valid configuration, one can choose a component $C \in \omega$ to serve the requests that come from a transaction T only if C has already an incoming past edge labeled with $root(T)$, and let the components in ω' serve the others. This means that components in ω cannot have incoming past edges labelled with new root transactions, and these components will eventually become free since old transactions will reach a stage where they will not use components in ω anymore. As soon as these components become free, they can be safely removed from the system.

However, if multiple versions cannot coexist, or it is preferable not to do it, the third strategy is called *blocking for freeness (BF)*. It requires that some of the requests to the components in ω be temporally blocked to avoid creating past edges labelled with new root transactions. These components will eventually become free when all existing transactions do not use them anymore. More precisely, the initiation of any transaction on any component in ω is blocked unless it belongs to an extended transaction set in which a member transaction has already been hosted by a component in ω. This comes from the fact that the first T-labelled past edge entering a component $C \in \omega$ is created the first time a transaction in $ext(T)$ is being initiated on C. As soon as all components in ω become free, they are substituted by the new versions in ω', and all blocked transactions are resumed.

Back to our example, if Auth receives the request for update before T_0 on Portal initiates T_1 on Auth, the initiation of T_1 is blocked because there is no T_0-labelled past edge entering Auth. Note that there can be T_0-labelled future edges entering Auth (e.g., see Fig. 3(a)), but by blocking the initiation of any $T \in ext(T_0)$ on Auth, the freeness of this component will not be hindered by T_0 since no T_0-labelled past edge needs to be created. However, if the request for update arrived at time ©, the initiation of T_3 by T_2 on Proc would be allowed because there is already a T_0-labelled past

edge entering Auth (created when T_1 ended). In this case, Auth must wait all T_0-labelled future edges to be removed, that is, at time Ⓓ, which corresponds to the configuration presented in Fig. 3(d).

This last strategy relies on the assumption that executing transactions are independent of each other. If they are not, that is, the progress of a transaction in an extended transaction set can be blocked by a transaction in another extended transaction set, we may have deadlocks. For example, this happens when two transactions are concurrently hosted on the same component and need to access a shared resource protected by a mutual exclusion lock. If one blocked the transaction that holds the lock (to avoid creating new past edges to the component-to-update), the other would be blocked as well (since it does not hold the lock), blocks would never be released, and the system would enter a deadlock.

However we do not expect this situation to be very common because modern systems often use fine grained locks to avoid blocking a transaction to serve other business-irrelevant transactions. Even when this situation does occur, remedies are possible. One can rely on existing mechanisms for the avoidance, prevention, or detection of distributed deadlocks. For example, according to the Wait-for Graph used for deadlock detection [18], one can resume the initialization of blocked transactions if their initializers are waited for by some transactions that must proceed to free the targeted components. Note that such a mechanism is very likely already provided by the infrastructure since distributed deadlocks are an important, potential problem in distributed systems with shared resources protected by coarse-grained locks. One can also use a more conservative blocking policy, and delay the creation of future edges —instead of past edges— labelled with new root transactions. This method prevents the initialization of any new root transaction that may use the targeted component, thus introducing more disruption. In practice, one may first take an optimistic approach by ignoring the deadlock problem and exploit these remedies when the freeness is not achieved after a reasonable time interval.

Generally, CV is preferred when applicable. WF and CV do not introduce disruption to system's service other than that caused by setting up and maintaining the valid configuration, while BF imposes extra disruption due to its temporal blocking of some transactions. As for timeliness, CV and BF strategies are essentially equivalent.

5. SIMULATIONS

Besides ensuring safety, dynamic reconfiguration is expected to be timely and to cause limited disruption. This is why the objective of this section is to assess the timeliness and degree of disruption of our approach and to compare them with those offered by the quiescence-based solution. We also aim to evaluate the impact of different levels of network latency on both approaches. The tranquillity-based approach was not included in the comparisons because it only ensures a local consistency property while the two approaches above ensure global consistency. Timeliness is measured as the time span between receiving a request for dynamic reconfiguration and entering a state where the system is ready for the changes. The degree of disruption is measured as the loss of working time for business transactions with respect to the execution of these transactions without dynamic reconfiguration.

Figure 4 shows the framework we used for the experiments[4]. To make our simulation realistic, the topologies of the systems we wanted to simulate are directed scale-free graphs[5] randomly generated by means of the JUNG framework[6]. The behavior of a CBDS is mimicked through a discrete event simulator based on the DEUS framework [2]. Events from the simulator drive the management framework to maintain the configuration according to the algorithm described in the previous section. Configurations are maintained on demand, and components are assumed to have accurate $f(T)$ and $p(T)$ for each locally-hosted ongoing transaction T, but no information other than static dependences for transactions that have already completed or have not started yet. The quiescence-based approach is implemented on top of the same framework by recursively passivating all components that statically depend on the to-be-updated ones.

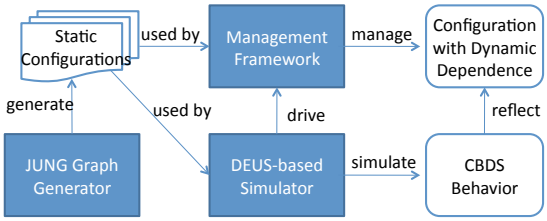

Figure 4: Simulation framework.

We also injected root transactions in all components with an activation pattern that satisfies Poisson distribution. Local processing times of transactions (not including the processing time of their sub-transactions) are normally distributed within a range of $(0, 2\mu)$ with a standard deviation of $\mu/5$, where μ is the mean local processing time. Transactions randomly initiate sub-transactions on the neighbor components of their host components. On average a transaction initiates one sub-transaction through each of its static outgoing edges. The actual system's workload for a particular configuration is given by the mean time interval between the activation of two root transactions at each component and the mean processing time of local transactions.

This framework allowed us to simulate systems with a different number of components and different workloads. The goal was to assess: (a) the impact of the system's size on the actual capabilities of ensuring version consistency, and (b) the performance of different strategies in achieving freeness with respect to different workloads.

5.1 Experiments

In our first experiment, we compared the timeliness and disruption of our approach[7] against those of the quiescence-based approach with systems of four different sizes: we generated 100 configurations of systems with 4, 8, 16, and 32

[4]The code of our experiments is available at: `http://vcsim.googlecode.com`.

[5]Scale-free graphs have been proposed as generic, universal models of network topologies that exhibit power law distributions in the connectivity of network nodes.

[6]`http://jung.sourceforge.net/index.html`.

[7]We used strategies CV and BF. For BF, the extended transaction sets are independent of each other as for liveness.

components. The mean processing time for local transactions was set to 50 time units[8], and the mean arrival interval between two root transactions at each component was set to 25 time units. The message delay between two neighbor components was set to 5 time units since the network delay between servers is often about an order of magnitude lower than the local processing time [14]. For each configuration, we randomly selected a component that had other components depending on it as the target for update, and probed the timeliness and disruption in the two cases.

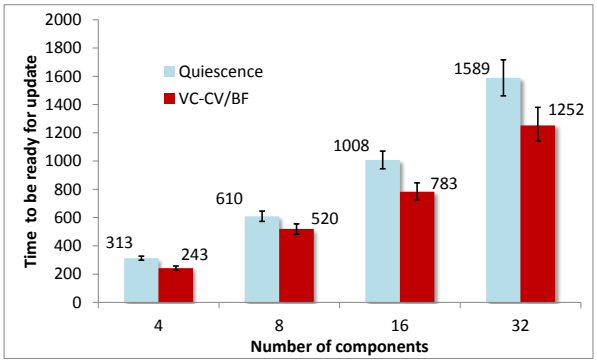

Figure 5: Timeliness of dynamic reconfiguration.

Figure 5 presents the mean timeliness of the two approaches and shows that freeness is achieved in a time that is some 20% less than the one needed to obtain quiescence.

Figure 6 shows that the degree of disruption of our approach with strategy BF is only about a half of that for the quiescence approach. With strategy CV, our approach only introduces negligible disruption, which is mainly due to the on-demand set up and maintenance of configurations. The standard errors of these values (shown by the error bars in the two figures) are relatively high, which indicates that the timeliness and disruption of both approaches are sensitive to the difference in the components targeted for update, the distribution of transactions on components, and the actual progressing of these transactions. However we observed that given the same configuration, even with strategy BF, our approach outperformed the quiescence-based one in 95% of the runs. These results then indicate that version-consistent dynamic reconfiguration, even with the overhead of handling dynamic edges introduced by the management algorithm, can be significantly less disruptive and more timely than the quiescence-based one.

The results also confirm the intuition that, for both ours and the quiescence-based approach, maintaining the consistency of dynamic reconfigurations becomes more and more expensive when the system scale increases.

Our second experiment was designed to evaluate the impact of network latency on the performance of the two approaches. In this experiment, we used the same settings introduced above with a couple of differences: we only show systems with 16 components —since the other configurations manifested the same trends— and message delay varied from 0 to 100. The results in Figure 7 show that our approach is consistently better than the quiescence-based one as far as

[8]Since the time is simulated, all time-related values in this section are in virtual time units unless otherwise specified.

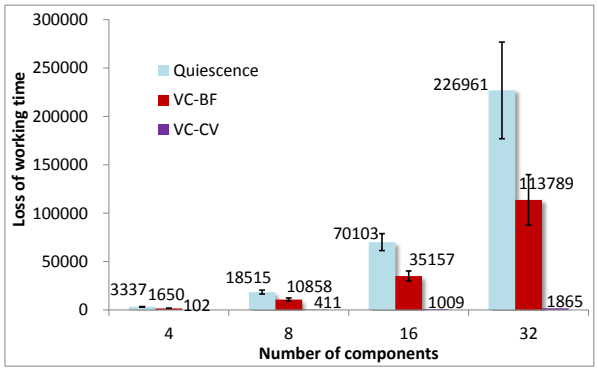

Figure 6: Disruption of dynamic reconfiguration.

disruption is concerned, but its gain in timeliness diminishes while message delay increases.

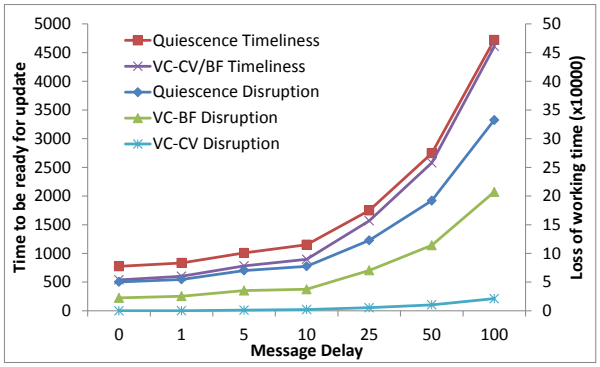

Figure 7: Impacts of network latency.

The third experiment presented here was aimed to analyze the impact of different strategies in achieving freeness on the timeliness of the actual reconfiguration. Especially, we want to see when strategy WF (simply based on waiting) can be applied and when we must use strategies CV (using concurrent versions) or BF (based on blocking). Intuitively the timeliness of strategy WF would be highly sensitive to the actual workload —more running transactions mean less opportunity for freeness. Figure 8 gives the time to freeness for strategies WF and CV/BF. The strategies were applied on the same 16-node configuration, and the same node was chosen as the target for update. The average local processing time was fixed to 50 time units, but with different intervals as for the arrival of root transactions, and thus different levels of workload. Figure 8 shows the average timeliness of 100 runs for each of the 9 different arrival intervals we selected (e.g., 1600 means that a new root transaction is initiated every 1600 time units). The results suggest that WF is preferable when the workload is rather light, but when it increases CV or BF ensures better timeliness.

6. RELATED WORK

The dynamic update of running applications has been extensively studied in multiple areas including programming languages [12, 13, 22, 23], operating systems [3, 10, 16], and software engineering [6,21]. A common theme of these works

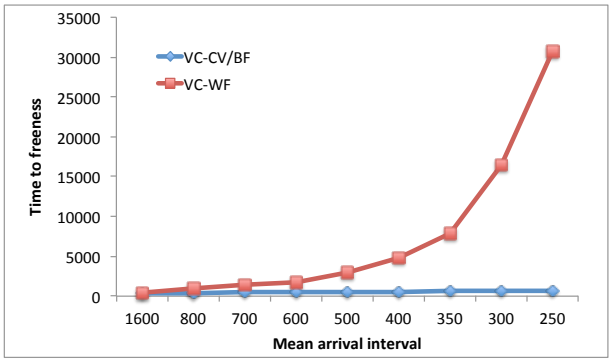

Figure 8: Timeliness: waiting vs. blocking.

is the selection of proper time points when the state of the system is steady and ready for applying a user-specified state transformation. The result is a new valid state from which the system is able to continue its evolution. Since generally the validity of the resulting state is undecidable [12], most research efforts focus on: (a) human-assisted identification of proper time points and state transformers by providing the necessary runtime support and (b) improving the timeliness of updates by automatically deriving further safe time points from those specified or known by the user.

Instead of switching from the old to the new version of the application, some works proposed intermediate versions to smooth the adaptation process. For example, Zhang and Cheng [27] proposed a model-based approach for the development of adaptive software. The behaviors of the different versions of an application are modeled by different state machines, and the adaptation behavior is rendered through states/transitions that connect them. Biyani and Kulkarni [5] used adaptation lattices to model transition paths from old to new programs, and introduced the concept of transitional-invariant lattice to verify the correctness of adaptations.

As for CBDSs, it is desirable and possible to avoid the direct manipulation of application-specific states of components and to maintain a clear separation of concerns between reconfiguration management and application logic. The focus is often on identifying suitable conditions (abstract states) under which components can be safely manipulated. The concept of quiescence by Kramer and Magee [15] is a prominent early work, but it may impose too high disruption on system's service. Subsequently, the dynamic reconfiguration service for CORBA by Bidan et al. [4], the proposal by Chen [7], and the idea of tranquillity by Vandewoude et al. [25] reduced disruption by means of considerations based on dynamic dependences, but they only guarantee some local consistency properties or impose stringent restrictions on the systems that can be updated.

It is also widely recognized that the dynamic adaptation of software systems should be modeled, analyzed, and managed at architectural level [9, 20]. Architectural models provide abstract global views of systems and explicitly specify system-level integrity constraints that must be preserved by reconfiguration. However, these models do not usually provide information about the runtime dynamic dependences needed to perform safe and low-disruptive runtime reconfigurations. Wermelinger [26] et al. proposed a category theory-based approach for the uniform modeling of the computations performed by components and their architectural configurations. This proposal can be used as a formal foundation for the specification of and reasoning about dynamic reconfigurations. However its complexity prevents it from being directly used in any concrete runtime management framework. Taentzer et al. [24] proposed the use of distributed graph transformation to model configurable distributed systems, but the dynamic dependences among components are not specified directly. We chose to add future and past edges to architectural configurations because they provide a simple but powerful abstraction for dynamic dependences, and provide a good separation of concerns between application-specific computation and reconfiguration management.

As the name suggests, our version consistency criterion is inspired by the work on transactional version consistency by Neamtiu et al. [19]. This work focuses on the dynamic update of centralized applications at code level, and its notion of transaction is a user-defined scope in the code. Their approach ensures that the execution of the code in the scope complies with the same, single version, no matter when the update happens. However, in our work the notion of version consistency is defined for a distributed system model. Neamtiu et al.'s approach [19] relies on global static program analysis and cannot be used in a distributed setting. Moreover, we use the notion of version consistency as a sufficient condition for the correctness of dynamic reconfigurations, while they use it as a user-specified pre-requisite for dynamic updates.

7. CONCLUSIONS AND FUTURE WORK

Dynamic reconfiguration is widely desired but its application in practice is still limited, partly because of the complexity in balancing consistency (of changes) and disruption (of system's service). This is why we propose the use of version consistency as a criterion for safe dynamic reconfigurations of CBDSs. The approach does not compromise the correctness of distributed transactions, but it allows for better timeliness and lower disruption than previous approaches. The paper also proposes a distributed mechanism to manage dynamic dependences between components at runtime, and thus to support version consistent dynamic reconfigurations.

As for our on-going and future work, we are working on an implementation of our framework on top of a BPEL engine. We also have plans to explore to what extent our approach can be paired with other optimizations of dynamic reconfigurations, such as supporting the coordinated abortion of re-triable transactions.

Acknowledgment

This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977 SM-Scom. Xiaoxing Ma is also partially supported by the NSFC (60973044, 60736015, 61021062) and the 973 Program of China (2009CB320702).

8. REFERENCES

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[2] M. Amoretti, M. Agosti, and F. Zanichelli. DEUS: a discrete event universal simulator. In *Simutools '09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1–9, ICST, Brussels, Belgium, Belgium, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[3] A. Baumann, G. Heiser, J. Appavoo, D. D. Silva, O. Krieger, R. W. Wisniewski, and J. Kerr. Providing dynamic update in an operating system. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 32–32, Berkeley, CA, USA, 2005. USENIX Association.

[4] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for CORBA. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 35, Washington, DC, USA, 1998. IEEE Computer Society.

[5] K. N. Biyani and S. S. Kulkarni. Assurance of dynamic adaptation in distributed systems. *J. Parallel Distrib. Comput.*, 68(8):1097–1112, 2008.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. POLUS: A powerful live updating system. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 271–281, 2007.

[7] X. Chen and M. Simons. A component framework for dynamic reconfiguration of distributed systems. In *CD '02: Proceedings of the IFIP/ACM Working Conference on Component Deployment*, pages 82–96, London, UK, 2002. Springer-Verlag.

[8] J. E. Cook and J. A. Dage. Highly reliable upgrading of components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*, pages 203–212, New York, NY, USA, 1999. ACM.

[9] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[10] C. Giuffrida and A. S. Tanenbaum. Cooperative update: a new model for dependable live update. In *HotSWUp '09: Proceedings of the Second International Workshop on Hot Topics in Software Upgrades*, pages 1–6, New York, NY, USA, 2009. ACM.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[12] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, 1996.

[13] M. Hicks and S. Nettles. Dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 27(6):1049–1096, 2005.

[14] U. Hoelzle and L. A. Barroso. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 2009.

[15] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[16] K. Makris and K. D. Ryu. Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 327–340, New York, NY, USA, 2007. ACM.

[17] X. Ma, L. Baresi, C. Ghezzi, V. Panzica La Manna and J. Lu. Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems. Technical Report, 2010. Available at http://home.dei.polimi.it/baresi/docs/vcdu-TR.pdf

[18] D. P. Mitchell and M. J. Merritt. A distributed algorithm for deadlock detection and resolution. In *Proceedings of the third annual ACM symposium on Principles of distributed computing 1984 (PODC '84)*, pages , 282–284, New York, NY, USA, 1984. ACM.

[19] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 37–49, 2008.

[20] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 899–910, New York, NY, USA, 2008. ACM.

[21] S. C. Previtali. Dynamic updates: another middleware service? In *MAI '07: Proceedings of the 1st workshop on Middleware-application interaction*, pages 49–54,2007.

[22] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. Mutatis mutandis: Safe and predictable dynamic software updating. *ACM Trans. Program. Lang. Syst.*, 29(4):22, August 2007.

[23] S. Subramanian, M. Hicks, and K. S. McKinley. Dynamic software updates: a vm-centric approach. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 1–12.

[24] G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change management by distributed graph transformation: Towards configurable distributed systems. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 179–193, London, UK, 2000. Springer-Verlag.

[25] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, 2007.

[26] M. Wermelinger, A. Lopes, and J. L. Fiadeiro. A graph based architectural (re)configuration language. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 21–32.

[27] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, 2006.