Version-consistent Dynamic Reconfiguration of Component-based Distributed Systems *

Xiaoxing Ma[1,2], Luciano Baresi[1], Carlo Ghezzi[1], Valerio Panzica La Manna[1], and Jian Lu[2]

[1]*Dipartimento di Elettronica e Informazione, Politecnico di Milano, 20133 Milano, Italy*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, 210093 Nanjing, China*
Email: {ma,baresi,ghezzi,panzica}@elet.polimi.it, lj@nju.edu.cn

Abstract

In these days there is an increasing demand for the runtime reconfiguration of distributed systems in response to changing environments and evolving requirements. Since these reconfigurations must be done in a safe and low-disruptive way, this paper proposes version consistency of distributed transactions as a safe criterion for dynamic reconfiguration. Version consistency ensures that distributed transactions be served as if there were 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 the architectural level and enable low-disruptive version-consistent dynamic reconfigurations. An initial assessment through simulation helped us evaluate the timeliness and degree of disruption of the proposed approach.

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 evolving requirements to satisfy. All these situations cannot be fully predicted at design time: some cases could be unknown, while others could be too expensive to identify and embed in the system from the beginning. For example, we may need to fix potential bugs, improve QoS, or add new functionality at runtime without blocking the system.

*This research has been partially funded by the European Commission, Programme IDEAS-ERC, Project 227977 SMScom. Xiaoxing Ma is also partially supported by the National Science Foundation of China under grant No. 60973044, 60736015, 60721002. Last revision: November 18, 2010.

Compared to off-line maintenance, dynamic modifications are more difficult since in addition to the correctness of the new version, they must also preserve 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 (*timeliness*).

Dynamic reconfigurations (of CBDSs) are not trivial [14, 23, 25]. At the architectural level, one generally avoids the direct manipulation of application-specific states, and drives the components targeted for reconfiguration into a *quiescent* status [14] before being reconfigured. 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 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. More transactions are allowed to run during reconfiguration, thus reducing the degree of disruption and improving the timeliness of the reconfiguration.

Existing proposals ([3, 7, 23]) 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 (or a set of components) of the system 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 accordingly 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 two representative approaches and paves the ground to the our proposal. Section 4 presents version consistency and the management framework. Section 5 discusses some practical issues needed to be considered when applying our approach. Section 6 summarizes the main results of the assessment conducted to evaluate the proposal. Section 7 surveys related approaches and Section 8 concludes the paper.

2 Problem setting

2.1 Component-based distributed system

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 *configuration*. Each node is tagged with the current version of the component it embodies. A valid configuration associates every out-port with exactly one in-port. A directed link from node N_i to node N_j means that N_i can send messages to N_j, and N_j can reply to N_i. Moreover, N_i is said to statically depend on N_j.

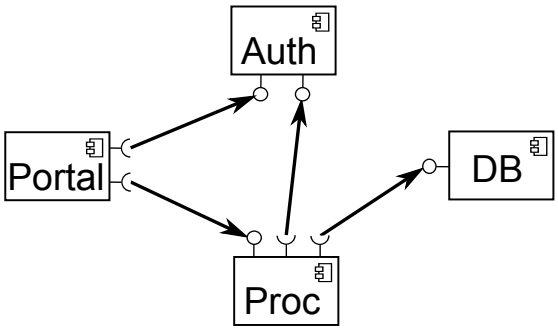

Figure 1: Our example system.

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.

A component can host (execute) transactions. A transaction is a sequence of actions that completes in bounded time. Actions include local computations and message exchanges. 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. The host component of transaction T is denoted as h_T. A transaction can initiate a sub-transaction on a neighbor component only when the host component of the initiating transaction statically depends on the neighbor component in charge of executing the new transaction. Transactions are also always notified of the completion of their sub-transactions. This implies that a transaction T cannot end before its sub-transactions T_i. All other exchanged messages between h_T and h_{T_i} —because of T_i— are temporally scoped between the two corresponding messages that initiate the sub-transaction and notify its completion.

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. Transactions are regarded as independent of each other unless they belong to the same extended transaction set. This allows us to state that the correctness of a transaction only depends on the correctness of its sub-transactions and the consistency of its host component (safety). Moreover, the progress of the transactions in an extended transaction set cannot be blocked by transactions in other extended transaction sets (liveness).

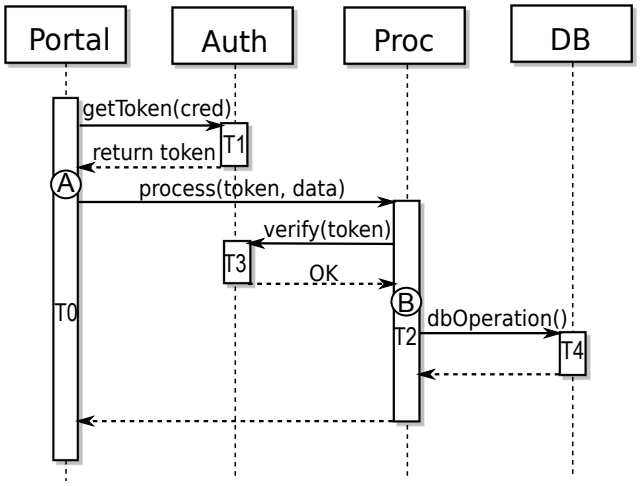

Figure 2: Detailed scenario.

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 services 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 getTocken 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.

2.2 Runtime updates

A dynamic reconfiguration defines how to drive the system into a suitable state and how to perform the update atomically, letting the system continue to work correctly. Dynamic reconfigurations are usually specified as atomic (sequences of) runtime updates of components with new versions. These updates are the most common reconfigurations in practice, and they can also be seen as the basic steps of complex reconfigurations [4, 25].

More rigorously, an atomic runtime 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 the state of the system with configuration $\Sigma' = \Sigma[\omega'/\omega]$[1]. The system

[1]This is to say that Σ' is equal to Σ where all components in ω have been substituted by

is expected to continue from s' with no externally-visible erroneous behaviors.

To focus on *dynamic* reconfiguration, we assume that given a runtime update $< \Sigma, \omega, \omega', \mathcal{T}, s >$, the corresponding off-line update $< \Sigma, \omega, \omega' >$ is correct. The *correctness* of the off-line update 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, and the absence of a single clock, we can only adopt a *weak* definition of correctness: a runtime update is correct iff:

- 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 suppose that Auth be updated to exploit a stronger encryption algorithm, and prevent weaknesses in system security. 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 root 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 be able to 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 such a correctness. One can view a transaction running on a component targeted for update as a process of the program of this component. Gupta et al. proved that given arbitrary program Π, new version of this program Π', state transformer $\mathcal{T}$ and valid state s of a process of Π, it is generally undecidable if the state $s' = \mathcal{T}(s)$ is a valid state of a process of Π' [11]. In addition to this undecidability, from an engineering perspective it is also undesirable to directly manipulate the internal temporal states of a component.

An obvious restriction on *when* the update can happen would be to impose that components targeted for update be *idle*. A component C is *idle* under system state s, denoted as $idle(s(C))$, means that it is not hosting transactions, and its *local* state $s(C)$ is equivalent[2] to the initial one $s_0(C)$. Unfortunately, this limitation is often insufficient for safe runtime updates. In fact, if we consider the scenario in Fig. 2, and substitute Auth when idle, but after serving getTocken, the resulting system would behave incorrectly since the security token would be created with an algorithm and validated by another.

Generally, we cannot have an automatically checkable condition sufficient and necessary for the correctness of an arbitrary on-line update, even if the corresponding off-line update is guaranteed to be correct and the on-line update happens when the component(s) to be updated is(are) idle.

Proposition 1. *Given arbitrary on-line update* $\langle \Sigma, \omega, \omega', \mathcal{T}, s \rangle$ *such that the corresponding off-line update* $\langle \Sigma, \omega, \omega' \rangle$ *is correct and* $\forall_{C \in \omega} idle(s(C))$*, it is undecidable whether the on-line update is correct or not.*

---

the new versions in ω'.

[2]This equivalence is subject to application-dependent interpretation. However at least it should be reflexive. This fact is used in the following justification for the undecidability of correctness of on-line update.

Proof. To show the undecidability, consider the following case. Assume the system consists of two components A and B. A provides a function f whose implementation uses a function g provided by B. The specification of the system is that $A.f()$ returns 0.

```
Component A:  int f() { return B.g(B.g(2)); }
Component B:  int g(int x) { return 0; }
```

Now component B is going to be updated on-line with B' that provides g'. The new system specification is the same as the old one. Suppose there is an algorithm $\mathcal{D}$ that can decide, for an arbitrary g' that ensures that the off-line update is correct, whether the on-line update that happens when B is idle is correct or not. Then we can use $\mathcal{D}$ to decide whether an arbitrary program $h()$ will eventually halt or not as follows. Let's construct g' as

```
int g'(int x) {
    switch (x){
      case 2: return 1;
      case 1: return 0;
      default: { h(); return 0; };
    }
}
```

Note that this g' ensures the correctness of the off-line update because $B'.g'(B'.g'(2))$ returns 0. Consider the situation that the update happens after $A.f$ calls $B.g$ in the first time but before the second time. Also note that in this moment, B is idle if the transaction is the only one in the system. In this situation $\mathcal{D}$ must be able to decide whether $B'.g'(B.g(2))$ returns 0. Since $B.g(2)$ returns 0, the result of $\mathcal{D}(g')$ is the same as whether $h()$ will eventually halt or not. Because whether an arbitrary program will eventually halt or not is undecidable, such an algorithm $\mathcal{D}$ can not exists. So the correctness of on-line update is generally undecidable because at least it is undecidable in this special case. □

The component computation and interaction mechanisms involved in the above argument for undecidability are very basic and it's unlikely that a practical component model would exclude them. So we can only derive some automatically checkable conditions that are sufficient for the correctness of runtime update by scoping the class of updates we consider. For any runtime update $\langle \Sigma, \omega, \omega', \mathcal{T}, s \rangle$, Σ and ω are given and the behavior of ω' is application-dependent and hard to check automatically. Moreover, since we have already imposed that to-be-updated components must be idle (each $C \in \omega$ is idle in s), $\mathcal{T}$ becomes useless[3], and we can only predicate on the system state s to identify the aforementioned sufficient conditions. Moreover, these conditions must be:

1. strong enough to ensure the correctness of on-line updates,
2. weak enough to allow for low-disruptive and timely changes, and
3. automatically checkable in a distributed setting (without enforcing unnatural centralized solutions).

[3]It would simply map the idle states of components in ω onto the idle states of components in ω'.

3 Quiescence and Tranquillity

In a seminal paper, Kramer and Magee [14] proposed a criterion called *quiescence* as sufficient condition for a node to be safely manipulated in dynamic reconfigurations.

Also their approach models a distributed system as a directed graph. A node can initiate transactions on itself, or initiate two-party transactions on another node if there is an edge between the two nodes. A node's state can only be affected by transactions. Every two-party transaction is a sequence of message exchanges between the two nodes. A (dependent) transaction T can "contain" other (consequent) transactions T_i: the completion of T depends on the completion of all the T_i. Transactions always complete in bounded time and the initiator is always notified about their termination.

Definition 1 (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 initiated by Portal and Proc. 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. This means that the faithful adoption of this approach could introduce significant disruption in the service provided by the system.

To reduce disruption, Vandewoude et al. [23] 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 anymore, even if the node has been involved in the transaction. Symmetrically, 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 2 (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" [23], the notion of tranquillity is based on a rather strong assumption. According to the definition of transaction of Section 2, given a root transaction T initiated at node N, $ext(T)$ can only contain sub-transactions hosted by nodes directly connected to N. This means that a sub-sub-transaction, hosted by a node that is not directly connected to N, is not part of the distributed transaction, and thus it can use any version of the components since it is an independent entity. For example, this criterion is not applicable to 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. Before the request for verification is sent, Auth has not participated in the session initiated by Proc. So Auth is tranquil at time Ⓐ. However, if Auth is updated at this time, the verification would fail because the token was issued by the old version of Auth with an incompatible encryption algorithm. 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. Moreover, the notation of transaction used by the two approaches is a bit too "narrow". Our notion of (distributed) transaction may involve more than two parties, and the dependency between a root transaction and its sub-transactions does not only refer to the termination of the different parts, but it may also affect their behavior.

4 Version-consistent Dynamic Reconfiguration

4.1 Version consistency

The following property states the *version consistency* criterion for legitimate dynamic reconfiguration.

Definition 3 (Version Consistency)**.** *Transaction T is version consistent iff $\nexists T_1, T_2 \in ext(T) \mid h_{T_1} \in \omega \wedge h_{T_2} \in \omega'$. A dynamic reconfiguration of a system is version consistent if all its transactions are kept version consistent.*

This means that a dynamic reconfiguration of a system is correct if it happens at such a time instant that all its transactions, including those started before and ended after the update, are kept version consistent. This is because of the correctness of the old and new configurations[4] and the fact that any version-consistent transaction is served—along with all its sub-transactions—as if it entirely completed within the old or the new configuration, no matter when the update actually happens. Also note that a transaction that ends before (starts after) the update cannot have a direct or indirect sub-transaction hosted by the new (old) version of a component being updated.

[4]Since we assume that the original system is correct, and that any off-line modification would produce another correct configuration of the system.

For our example, if the update of Auth happens after transaction T_0 begins but before it sends a getToken request to Auth, all transactions in $ext(T_0)$ (i.e., all transactions in Fig. 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 $h_{T_1} = \mathsf{Auth}$, but $h_{T_3} = \mathsf{Auth}'$. 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.

4.2 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 that ensures 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 Fig. 1 through properly-labelled edges besides those that represent the static dependencies. A static-labelled edge represents both a static dependence and the communication channel between the two components; future and past edges represent dynamic dependences. 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'. This means that because of T, some transactions in $ext(T)$ hosted by C will use (has used) the service provided by C' by initiating sub-transactions on it.

Definition 4 (Valid Configuration)**.** *A valid configuration with dynamic dependences, hereafter configuration, must satisfy the following constraints:*

1. (HOST-VALIDITY) *Given a transaction T, the hosting component C must have a dedicated pair of edges $C \xrightarrow[root(T)]{future} C$ and $C \xrightarrow[root(T)]{past} C$ during the whole duration of T.*
2. (LOCALITY) *Each future or past edge $C \xrightarrow[T]{future(past)} C'$ must be paired with a static edge $C \xrightarrow{static} C'$;*
3. (FUTURE-VALIDITY) *A future edge $C \xrightarrow[T]{future} C'$ begins to exist no later than the first time a transaction $T' \in ext(T)$ is initiated, and continues to exist at least until no transactions hosted by C will initiate further $T'' \in ext(T)$ on C' through $C \xrightarrow{static} C'$;*
4. (PAST-VALIDITY) *A past edge $C \xrightarrow[T]{past} C'$ begins to exist no later than the*

end of any transaction $T' \in ext(T)$ *initiated by a transaction hosted by* C *on* C' *through* $C \xrightarrow{static} C'$*, and continues to exist at least until the end of* T*.*

Fig. 3 shows some configurations of the example system where dynamic dependences are rendered explicitly (two concentric cycles correspond to a pair of local future/past edges). Fig. 3(a) describes the case in which transaction T_0 has begun at Portal and will use Auth and Proc. Note that to serve T_0, Proc will also use Auth and DB, and thus the corresponding future edges are set. Fig. 3(b) says that T_1 ($T_1 \in ext(T_0)$) is currently running at Auth, but no further transaction in $ext(T_0)$ hosted by Portal will initiate a sub-transaction on Auth because of the removal of the future edge. Fig. 3(c), which corresponds to time instant Ⓐ in Fig. 2, indicates that Auth has hosted transaction T_1 initiated by Portal in the past, and will possibly host other transactions in $ext(T_0)$ initiated by Proc in the future. Fig. 3(d) corresponds to instant Ⓑ in Fig. 2 and shows that Auth, although it has hosted transactions in $ext(T_0)$, is not hosting and will not host them anymore.

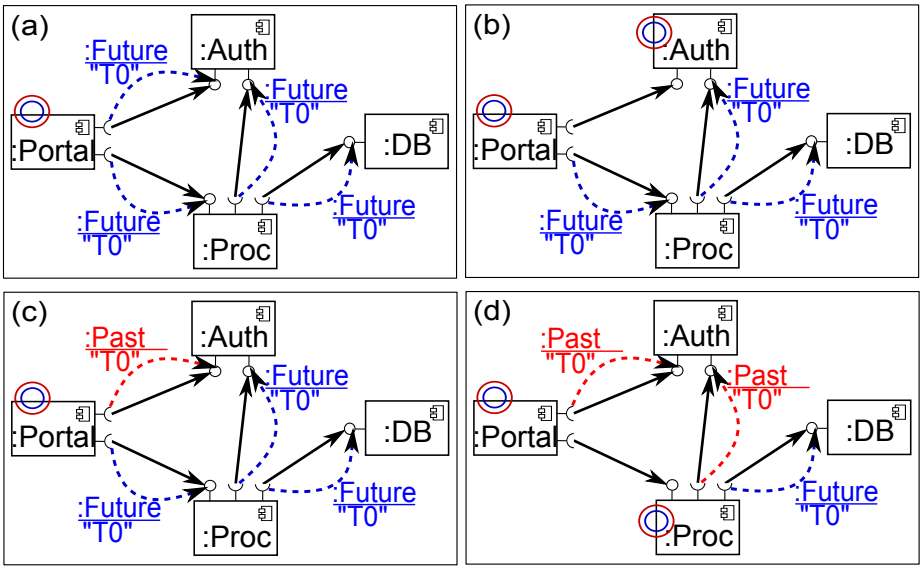

Figure 3: Some configurations of the example system with explicit dynamic dependences.

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

Definition 5 (Freeness)**.** *Given a configuration* G*, 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 future/past edges labelled with* T *arriving at* C *(*ω*). *C *(*ω*) is said to be free in* G *iff it is free with respect to all the transactions in the configuration.*

Auth is free of dependency with respect to T_0 in the configurations of Fig. 3(a) and 3(d), while the local dynamic edges at Auth trivially falsify its freeness in the

configuration of Fig. 3(b). Intuitively, for a valid configuration G, 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 2. *Given a valid configuration G of a system, a dynamic reconfiguration of a set of its components ω is version consistent if it happens when ω is free in G.*

Proof. If we suppose that the version consistency property 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 G and the freeness of 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 ($h_{T_1} \in \omega$), T begins before the update. Similarly T ends after the update because of T_2. Since no transaction is hosted by 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, 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 G 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 hosted by a component in ω' in $ext(T)$. According to the future-validity of G 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 arriving at components in ω. This contradicts the freeness of ω. □

4.3 Distributed management of dependences

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

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 these edges at the beginning of a root transaction and by removing them at the ending

of it. However, although version consistency would be ensured, disruption could be too high. As already shown in our example, optimizations are possible with some information about the application logic and about the progresses of local transactions.

We assume that given a transaction T, its host component h_T knows $f(T)$, the set of out-going static edges through which it might initiate sub-transactions on neighbor components in the future, and $p(T)$, the set of out-going static edges through which it has initiated sub-transactions in the past[5].

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 arriving at it. To keep the consistency of views among neighboring components, management messages are exchanged in addition to business (transaction-related) messages. 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.

The managements of dynamic edges for different distributed transactions are 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).

More in detail, the three steps carry out the following actions and they help preserve future and past validity as follows:

Setting 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 all these acknowledgements arrive, 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 back C'. 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

[5]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 in deciding the actual tranquillity of components [23].

can be handled by avoiding creation and notification of duplicated future edges.

This conservative way creates *enough* future edges to achieve a valid configuration with respect to T when the set up step finishes. Fig. 3(a) shows the result of setting up future edges for transaction T_0 of Fig. 2.

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

More in detail, new *will-not-use* information can be available at various time points 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.

Again, 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 (Fig. 3(b)). When T_1 ends, Portal immediately creates a past edge to record the fact that it has used Auth (Fig. 3(c)). Eventually, the system reaches the configuration of Fig. 3(d), where Auth is free with respect to T_0.

Cleaning 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 is 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. The algorithm is presented in Appendix A.

4.4 Achieving freeness

With a valid configuration maintained at runtime, checking for the freeness of ω —the part of the system targeted for update— is straightforward since the

condition is local to ω. What is still unsolved, however, is the actual reachability of freeness. By following an opportunistic approach, the system may simply wait for freeness to manifest itself, but 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.

The above approach to maintaining validity of configurations provides a convenient way to achieve the freeness of ω. Under the assumption that transactions belonging to different extended transaction sets are independent of each other, we can just delay the creation of past edges labelled with the identifier of a new root transaction that does not appear in any of the existing past edges towards components in ω. This happens at the creation of a local past edge when some transactions are being initiated on the components in ω. The delay blocks the initiation of any transaction on any component in ω unless it belongs to an extended transaction set in which a member transaction has already been hosted by some components in ω.

For our example, if Auth receives the update request before T_0 at Portal initiates T_1 on Auth, the initiation of T_1 will be blocked because there is no T_0-labelled past edge arriving at Auth. Note that at this time there can be T_0-labelled future edges arriving at Auth (see Fig. 3(a)), but by blocking the initiation of any $T \in ext(T_0)$ on Auth the freeness of this component will no be hindered by T_0 since no T_0-labelled past edge needs to be created. However, if the updated request arrives at time point Ⓐ in Fig. 2, the initiation of T_3 by T_2 at Proc will be allowed because there is already a T_0-labelled past edge arriving at Auth created when T_1 ended. In this case Auth must wait all T_0-labelled future edges to be removed (at time point Ⓑ in Fig. 2 and also in Fig. 3(d)).

5 Practical issues

The actual application of our proposal requires that some further practical issues be considered. In this section we discuss how to use version consistency scopes and on-demand configuration setting up to minimize the overhead of our proposal. We also further examine our assumption of independence between different distributed transactions and discuss some possible solutions for the problem arisen when the assumption is broken.

5.1 Version consistency scope

Keeping version consistency over the whole system can be expensive and unnecessary. Users are allowed to strike a balance for their needs by explicitly specifying the scope of "their" version consistency. The version consistency scope $\mathcal{S}$ is a part of the system configuration, which contains the part targeted for update. For simplicity from now we just regard it as a set of components. Let $sub_{\mathcal{S}}(u, v) = sub(u, v) \wedge h_u, h_v \in \mathcal{S}$ be the sub-transaction relationship scoped in $\mathcal{S}$, and $ext_{\mathcal{S}}(T) = \{x | (x = T) \vee sub^{+}_{\mathcal{S}}(T, x)\}$ be the extended transaction set of T scoped in $\mathcal{S}$.

Definition 6 (Scoped Version Consistency)**.** *Transaction T is version consistent within a scope $\mathcal{S}$ iff $\nexists T_1, T_2 \in ext_{\mathcal{S}}(T) \mid h_{T_1} \in \omega \wedge h_{T_2} \in \omega'$. A dynamic reconfiguration is version consistent within a scope $\mathcal{S}$ iff all transactions are kept version consistent within $\mathcal{S}$.*

By specifying a version consistency scope for a runtime update, a user tells the configuration management that the impact of the update is known to be confined within this scope, that is, if viewed from the outside, the behavior of this part of the system does not change despite the update. For our example, one can specify the version consistency scope of {Auth, Portal, Proc} for the replacement of Auth. In large, loosely coupled systems, one could expect that only relatively small scopes are needed.

When the scope of version consistency $\mathcal{S}$ is specified, we only need to consider dynamic dependences within $\mathcal{S}$ by limiting the sub-transaction relationship within it. A configuration with dynamic dependences G is valid with respect to $sub_{\mathcal{S}}$ means that G would be valid if every transaction T were independent root transactions unless there existed T' such that $sub_{\mathcal{S}}(T', T)$. This implies that there is no dynamic edge out of $\mathcal{S}$ except for those of host-validity. Then Proposition 2 directly implies the following proposition.

Proposition 3. *Given a configuration with dynamic dependences G of a system, a dynamic reconfiguration is version consistent with scope $\mathcal{S}$ if (1) G is valid with respect to $sub_{\mathcal{S}}$, and (2) the update is applied when ω, which is the part of the system to be updated, is free in G.*

To maintain the validity of a configuration with dynamic dependences with respect to a scope $\mathcal{S}$, we only need to consider the components in $\mathcal{S}$. For each component in $\mathcal{S}$ we apply the essentially same algorithm as above. In this case dynamic edges are labelled with identifiers of *deputy* root transactions. A transaction T is called a deputy root transaction in $\mathcal{S}$ if $h_T \in \mathcal{S}$ and $\nexists T' \mid sub_{\mathcal{S}}(T', T)$.

Nevertheless, the approach to achieving freeness in sub-section 4.4 could cause deadlocks when scoped version consistency is used. For example, let us suppose that C_1 and $C_3 \in \omega$, but $C_2 \notin \mathcal{S}$. C_1 hosts T_1 which initiated T_2 on C_2. Now T_2 is going to initiate T_3 on C_3. Since T_3 will be treated as a new deputy root transaction independent of T_1, it will be blocked when C_3 tries to create the corresponding local past edge labelled with a new identifier ($T3$). T_3 will not be allowed to progress until reconfiguration is done, but reconfiguration cannot be done until the freeness of ω is reached. This freeness cannot be reached before T_1 ends, but T_1 is waiting for T_2, which in turn is waiting for T_3 to end. One solution is to label the dynamic edges with the identifiers of *real* root transactions in addition to the identifiers of deputy root transactions. While the latter is used when maintaining the scoped version of configuration validity, the former is used when deciding if the creation of a past edge can be blocked or not —it is not blocked if there is some past edge headed towards a component in ω with the same identifier of real root transaction.

5.2 On-demand set up of dynamic dependences

The algorithm in sub-section 4.3 assumes that configurations are always kept up-to-date no matter whether there is a request for dynamic reconfiguration. However, configuration management could bring considerable overhead to the system. This means that 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[6]:

[6]For simplicity, the static configuration is assumed to be acyclic.

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 *enough* future (past) edges toward the components that might host in the future (might have hosted in the past) transactions in $ext(T)$ in a way similar to the the setting up of future edges described in sub-section 4.3.

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 described in sub-section 4.4 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 algorithm for on-demand setting up of dynamic edges is included in Appendix B.

5.3 Independence between transactions

Some sort of independence between transactions is often assumed in work on dynamic reconfiguration of CBDSs [6, 14, 23], but it's implications are seldom clearly explained. Our approach also requires that transactions in different extended transaction sets are independent of each other. With this assumption, we can deal with safety and liveness as follows.

On the aspect of safety, we are assuming that the correctness of a transactions only dependents on the correctness of its sub-transactions and the *consistency* of its host component. Here the consistency of a component means that its state satisfies some application-specific invariants and that a correct transaction will turn its host component from a consistent state to another consistent state. To support concurrent transactions, components also needs to provide some kind of *isolation* such as *serializability* [10]. The reciprocal causation between consistency of components and the correctness of transactions makes them closed properties of the system. Our approach to runtime update allows transactions whose sub-transactions hosted by different versions of components to co-exist on a same component. The independence between these transactions ensures the consistency of the component and the correctness of the transactions.

On the aspect of liveness, we are assuming that the progress of a distributed transaction will not be blocked by other distributed transactions. When this assumption does not hold, for example two distributed transactions both need to access a shared object protected with a mutual exclusion lock, deadlocks may happen if we use the approach given in sub-section 4.4 to achieving freeness. For instance, consider a scenario that a transaction holding a lock is blocked when it initiates a sub-transaction on a component to be updated, and a transaction running on this component is waiting for the lock. Note that if shared

resources and concurrency control are used in the distributed system, it's likely that the problem of possible distributed deadlocks is already there even without dynamic reconfigurations. In this case our approach to achieving freeness does not introduce a new problem but introduces a new shared resource—the component(s) to be updated. Anyway, if we allow this kind of dependence between distributed transactions, following approaches can be adopted to achieve the freeness of components targeted for update:

- Instead of blocking the creation of the first past edge labelled with a new identifier, one can block the creation of any new root transactions. This approach is pessimistic and could bring more disruption. Since deadlocks are not expected to be common in practice, an optimization is to start with the original approach, and switch to the pessimistic one only when the freeness is still not achieved after a reasonable time span.
- If the system already provides a distributed deadlock detection mechanism, such as edge-chasing, we can stick to the original approach and detect deadlock when suspected. If a deadlock presents, allow those transactions involved in the deadlock to create corresponding past edges and initiate their sub-transactions on the components to be updated.
- If possible, let the old and new versions of component(s) co-exist in the system, and in this case no blocking is needed. With the information of dynamic dependences provided by our management framework, it's straight forward to ensure version consistency by letting the old version serve distributed transactions that has been served before, and the new version serve new distributed transactions.

6 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 approach. 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 ensures global consistency of distributed transactions. 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.

6.1 Simulation framework

Fig. 4 shows the framework we used for the experiments. To make our simulation realistic, the topology of the systems we wanted to simulate are directed scale-free graphs[7] randomly generated by means of the JUNG framework[8]. The behavior of a CBDS is mimicked through a discrete event simulator based on

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

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

the DEUS framework [1]. 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. 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 component.

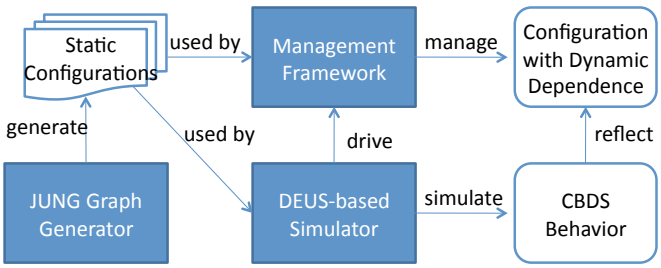

Figure 4: Simulation framework.

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 derivation 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 out-going 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 allows us to simulate systems with a number of components and actual workload. The goal is to assess the impact of the system's size on the actual capabilities of ensuring version consistency, and the performance of different strategies in achieving freeness with respect to different workloads.

6.2 Experiments and results

In our first experiment, we compared the timeliness and disruption of our approach with those of the quiescence-based approach with systems of four different sizes: we generated 100 configurations for systems with 4, 8, 16, and 32 components. The mean processing time for local transactions was set to 50 time units[9], 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 [13, p. 33]. For each configuration, we randomly selected a component that had other components depending on it as the target for update, and probe the timeliness and disruption by using both our approach and the quiescence-based one.

[9]Since time is simulated, these are virtual units.

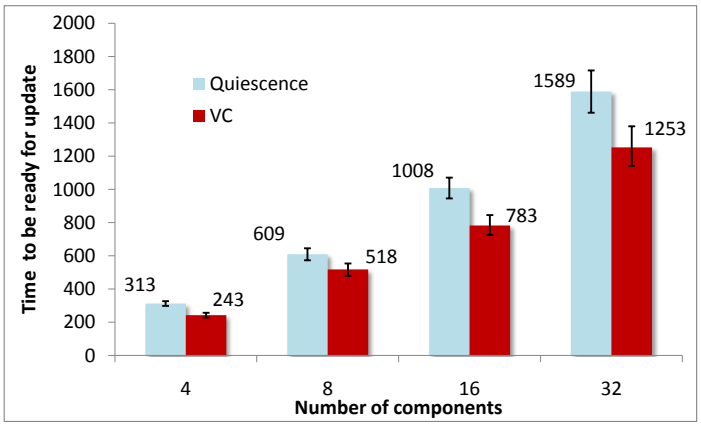

Figure 5: Timeliness of dynamic reconfiguration.

Fig. 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. Fig. 6 shows that the degree of disruption due to blocking for freeness (and also due to the on-demand set up of configurations) is only about a half of that due to passivating for quiescence. The standard errors of these values (shown with 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 target selected component, distribution of transactions on components, and actual progressing of these transactions. However we observed that given the same configuration, our approach outperformed the other in the 95% of the runs. These results then indicate that version-consistent dynamic reconfiguration can be significantly less disruptive and more timely than the quiescence-based one.

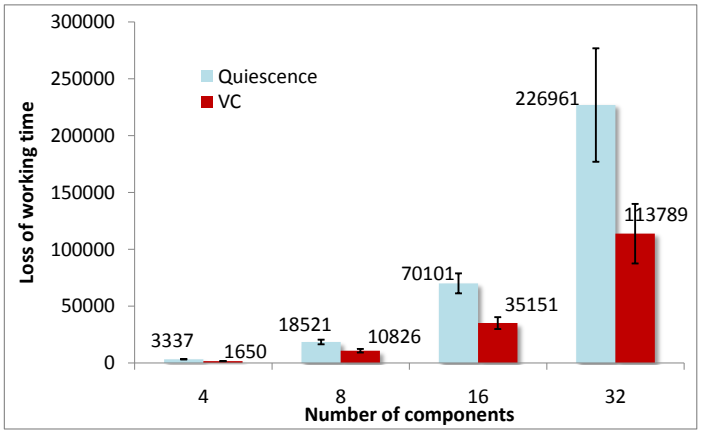

Figure 6: Disruption of dynamic reconfiguration.

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 used systems with 16 components, and message delay varied from 0 to 100. The results in Fig. 7 show that our approach is consistently better than the quiescence-based one as far as disruption is concerned, but its gain in timeliness diminishes while message delay increases.

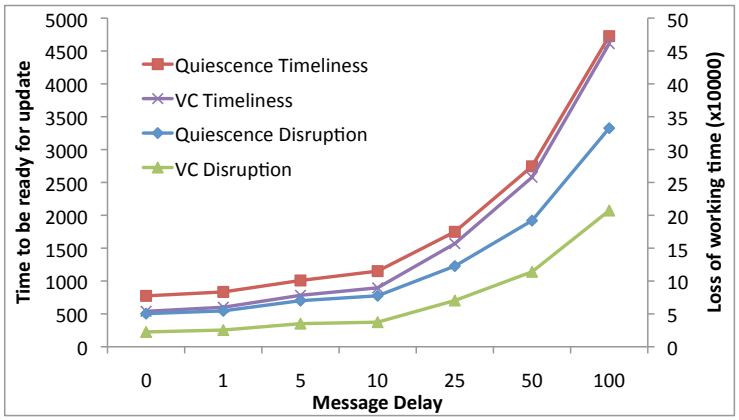

Figure 7: Impacts of network latency.

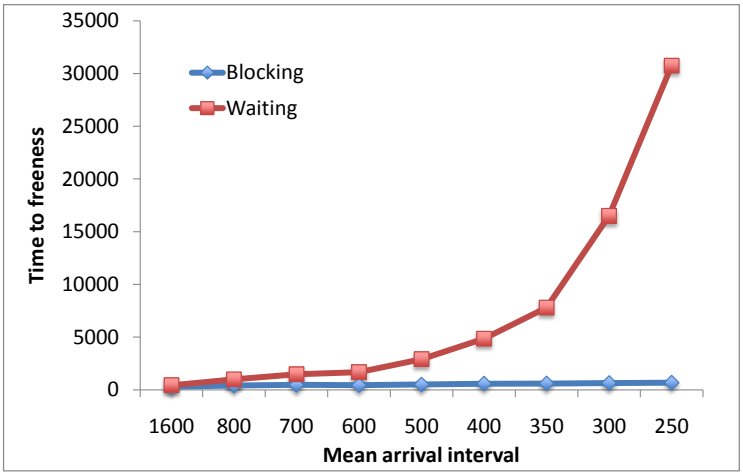

Figure 8: Timeliness: waiting vs. blocking.

The third experiment presented here was aimed to analyze the impact of different approaches in achieving freeness on the timeliness of the actual reconfiguration. In the above experiments, a transaction was blocked when initiating a sub-transaction on the component targeted for update unless the identifier of its root transaction appeared on some past edges entering the component. We

could also just wait for the component to become free by itself. Since no blocking is involved, the disruption caused by blocking is eliminated, but intuitively the timeliness of this approach would be highly sensitive to the actual workload —more running transactions mean less opportunity for freeness. Fig. 8 gives the time to freeness in the two cases: just waiting for freeness to appear and achieving freeness by blocking related transactions. The two approaches 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. Fig. 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 the former approach (simply based on waiting) is preferable when the workload is rather light, but when it increases the latter (based on blocking) ensures better timeliness.

7 Related Work

The dynamic update of running applications has been extensively studied in multiple areas including programming languages [11, 12, 20, 21], operating systems [2, 9, 15], and software engineering [5, 19]. A common theme of these works 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 [11], 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 [25] proposed a model-based approach for the development of adaptive software. The behavior of the different versions of an application are modeled with different state machines, and the adaptation behavior is rendered through states/transitions that connect them. Biyani and Kulkarni [4] 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 adaptation.

If we think of 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 statuses) under which components can be safely manipulated. The concept of quiescence by Kramer and Magee [14] is a prominent early work, but it imposes too high disruption on system's service. Subsequently, the blocking algorithm proposed by Moazami-Goudarzi and Kramer [16], the dynamic reconfiguration service for CORBA by Bidan et al. [3], the proposal by Chen [7], and the idea of tranquillity by Vandewoude et al. [23] reduced disruption by means of considerations based on dynamic dependences, but they only guarantee some local consistency prop-

erties or impose stringent restrictions on the systems to which the approaches can be applied.

It is also widely recognized that the dynamic adaptation of software systems should be modeled, analyzed, and managed at architectural level [8, 18]. 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 runtime dynamic dependences needed to perform safe and low-disruptive runtime reconfigurations. Wermelinger [24] 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 complete 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. [22] 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. [17]. This work focuses on the dynamic update of centralized applications at code level, and its notion of transaction is a user-specified 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 updated happens. One can think of architectural configurations of CBDSs as "programs", and distributed transactions as executions of these programs. Then our concept of version consistency is comparable to theirs. Their use of static program analysis techniques is also interesting to us. As part of our future work, we will investigate similar techniques to automatically generate accurate future and past edges from components' code.

8 Conclusions and Future Work

Dynamic reconfiguration is widely desired but its application in practice is still limited (partially) 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: it 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 a complete framework to support dynamic reconfiguration and on a two-layer graph transformation model that uniformly formalizes both the specification and management of system configurations with dynamic dependences. We also have plans to explore to what extent our approach can be paired with other optimizations of dynamic reconfigurations, such as using multiple concurrent versions of the part targeted

for update and supporting the coordinated abortion of retriable transactions.

References

[1] 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).

[2] 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.

[3] 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.

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

[5] 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, Washington, DC, USA, 2007. IEEE Computer Society.

[6] X. Chen. Dependence management for dynamic reconfiguration of component-based distributed systems. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 279, Washington, DC, USA, 2002. IEEE Computer Society.

[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] 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.

[9] 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.

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

[11] 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.

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

[13] 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.

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

[15] 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.

[16] K. Moazami-Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. In *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, page 62, Washington, DC, USA, 1996. IEEE Computer Society.

[17] 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, New York, NY, USA, 2008. ACM.

[18] 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.

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

[20] 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.

[21] 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, New York, NY, USA, 2009. ACM.

[22] 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.

[23] 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.

[24] 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, New York, NY, USA, 2001. ACM.

[25] 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, New York, NY, USA, 2006. ACM.

A Algorithm for dynamic dependence management

use $f(T)$ *{out-going static edges might be used in the future by T}*
use $p(T)$ *{out-going static edges used in the past by T}*
use $localTxs()$ *{on-going local transactions}*
const THIS *{this node}*
var $OES \leftarrow \{\}$ *{dynamic edges leaving from this node}*
var $IES \leftarrow \{\}$ *{dynamic edges arriving at this node}*

Setting up:

Upon being initiated a root tx T:
let $lfe = \text{THIS} \xrightarrow[T]{future} \text{THIS}$, $lpe = \text{THIS} \xrightarrow[T]{past} \text{THIS}$
$OES \leftarrow OES \cup \{lfe, lpe\}$, $IES \leftarrow IES \cup \{lfe, lpe\}$

Upon a root tx T initiating its first sub-tx:
for all $ose = \text{THIS} \xrightarrow{static} C \in f(T)$ **do**
  let $fe = \text{THIS} \xrightarrow[T]{future} C$
  $OES \leftarrow OES \cup \{fe\}$
  NOTIFY_FUTURE_CREAT fe
  wait for ACK_FUTURE_CREAT fe
end for

Upon receiving NOTIFY_FUTURE_CREAT $xe = C \xrightarrow[T]{future} \text{THIS}$
$IES \leftarrow IES \cup \{xe\}$
for all $C' \in \{X \mid \text{THIS} \xrightarrow{static} X\}$ **do**
  let $fe = \text{THIS} \xrightarrow[T]{future} C'$
  if $fe \notin OES$ **then**
    $OES \leftarrow OES \cup \{fe\}$
    NOTIFY_FUTURE_CREAT fe
    wait for ACK_FUTURE_CREAT fe
  end if

22: **end for**
23: ACK_FUTURE_CREAT xe

Progressing:

1: **sub** remove_future_edges T *{remove T-labelled future edges without breaking future-validity}*
2: **for all** $fe \in OES : fe = \text{THIS} \xrightarrow[T]{future} C \ \wedge \ C \neq \text{THIS}$ **do**
3: **if** $(\nexists e \in IES : e = C' \xrightarrow[T]{future} \text{THIS} \ \wedge \ C' \neq \text{THIS}) \wedge$
$(\nexists T' \in localTxs() : \ root(T') = T \ \wedge \ \text{THIS} \xrightarrow{static} C \ \in \ f(T'))$ **then**
4: $OES \leftarrow OES - \{fe\}$
5: NOTIFY_FUTUE_REMOVE fe
6: **end if**
7: **end for**
8: **end sub**
9:
10: *Upon being initiated a sub-tx T via $se = C \xrightarrow{static} \text{THIS}$:*
11: **let** $lfe = \text{THIS} \xrightarrow[root(T)]{future} \text{THIS}$, $lpe = \text{THIS} \xrightarrow[root(T)]{past} \text{THIS}$
12: $OES \leftarrow OES \cup \{lfe, lpe\}$, $IES \leftarrow IES \cup \{lfe, lpe\}$
13: ACK_SUBTX_INIT se $root(T)$
14:
15: *Upon receiving* ACK_SUBTX_INIT T *via* $se = \text{THIS} \xrightarrow{static} C$*:*
16: remove_future_edges T
17:
18: *Upon receiving* NOTIFY_FUTUE_REMOVE $fe = C \xrightarrow[T]{future} \text{THIS}$*:*
19: $IES \leftarrow IES - \{fe\}$
20: remove_future_edges T
21:
22: *Upon ending a sub-tx T initiated via $se = C \xrightarrow{static} \text{THIS}$:*
23: NOTIFY_SUBTX_END se $root(T)$
24:
25: *Upon receiving* NOTIFY_SUBTX_END T *via* $se = \text{THIS} \xrightarrow{static} C$*:*
26: **let** $pe = \text{THIS} \xrightarrow[T]{past} C$
27: $OES \leftarrow OES \cup \{pe\}$
28: NOTIFY_PAST_CREATE pe
29:
30: *Upon receiving* NOTIFY_PAST_CREATE $pe = C \xrightarrow[T]{past} \text{THIS}$*:*
31: $IES \leftarrow IES \cup \{pe\}$
32: **if** $\nexists T' \in localTxs() : root(T') = T$ **then**
33: **let** $lfe = \text{THIS} \xrightarrow[T]{future} \text{THIS}$, $lpe = \text{THIS} \xrightarrow[T]{past} \text{THIS}$
34: $OES \leftarrow OES - \{lfe, lpe\}$, $IES \leftarrow IES - \{lfe, lpe\}$
35: **end if**
36: remove_future_edges T

Cleaning up:

1: **sub** remove_all_edges T *{remove all T-labelled edges}*
2: **for all** $e \in OES : e = \text{THIS} \xrightarrow[T]{*} *$ **do**

3: $OES \leftarrow OES - \{e\}$
4: **if** $e = \text{THIS} \xrightarrow[T]{future} C \wedge C \neq \text{THIS}$ **then**
5: NOTIFY_FUTURE_REMOVE e
6: **else if** $e = \text{THIS} \xrightarrow[T]{past} C \wedge C \neq \text{THIS}$ **then**
7: NOTIFY_PAST_REMOVE e
8: **end if**
9: **end for**
10: **end sub**
11:
12: *Upon ending a root tx T:*
13: **let** $lfe = \text{THIS} \xrightarrow[T]{future} \text{THIS}$, $lpe = \text{THIS} \xrightarrow[T]{past} \text{THIS}$
14: $OES \leftarrow OES - \{lfe, lpe\}$, $IES \leftarrow IES - \{lfe, lpe\}$
15: remove_all_edges T
16:
17: *Upon receiving* NOTIFY_PAST_REMOVE $pe = C \xrightarrow[T]{past} \text{THIS}$*:*
18: $IES \leftarrow IES - \{pe\}$
19: remove_all_edges T

To see why this algorithm keeps the validity of the configuration with dynamic dependences, one can observe that:

Locality Locality holds because a non-local future or past edges $\text{THIS} \xrightarrow[T]{future/past} C$ is only created when there is a static edge $\text{THIS} \xrightarrow{static} C$ (see line 8 and 18 of Setting up and line 27 of Progressing);

Host-validity For every root transaction, corresponding local dynamic edges are set up at the Setting up stage and removed at cleaning up stage. For a sub-transaction, corresponding local dynamic edges are set up upon initiation (line 12 of Progressing) if there are not such edges, and are only removed when there is no locally hosted on-going transaction belonging to the same extended transaction set (line 34 of Progressing).

Future-validity For a root transaction T, the Setting up step creates enough future edges labelled with T to all component might host $T' \in ext(T)$ in the future. In the Progressing step, a non-local future edge $fe = C \xrightarrow[T]{future} C'$ is removed only when (line 3 of Progressing): (1) no on-going $T' \in ext(T)$ hosted by C will initiate any sub-transaction on C' through $se = C \xrightarrow{static} C'$ any more; 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 any more. This condition ensures future validity because once fe is removed, no sub-transactions in $ext(T)$ need to be initiated through se any more. Other removal of future edges is in the Cleaning up step and does not affect future-validity.

Past-validity When a sub-transaction T (initiated by T' through $se = h_{T'} \xrightarrow{static} h_T$) ends, a corresponding past edge $pe = h_{T'} \xrightarrow[root(T)]{past} h_T$ is created before

the possible removal of the corresponding local edges at h_T, which effectively delay the ending of T until the corresponding past edge is created. The removal of past edges only happens in Cleaning up step when $root(T)$ ends. So the past-validity is ensured.

B Algorithm for on-demand set up of dynamic dependences

On-demand Setting up

use $f(T)$ *{out-going static edges might be used in the future by T}*
use $p(T)$ *{out-going static edges used in the past by T[10]}*
use $s(T)$ *{out-going static edges currently being used by T}*
use $localTxs()$ *{on-going local transactions}*
const THIS *{this node}*
var $OES \leftarrow \{\}$ *{dynamic edges leaving from this node}*
var $IES \leftarrow \{\}$ *{dynamic edges arriving at this node}*
var $mode \leftarrow$ NORMAL *{NORMAL, ONDEMAND, or VALID mode}*

1: *Upon receiving* REQ_ONDEMAND_SETUP $ose =$ THIS $\xrightarrow{static} C$
2: **while** have not received REQ_ONDEMAND_SETUP from every in-scope[11] outgoing static edge[12] **do**
3: wait for other REQ_ONDEMAND_SETUP requests
4: **end while**
5: $mode \leftarrow$ ONDEMAND
6: **for all** incoming static edge $ise = C' \xrightarrow{static}$ THIS **do**
7: REQ_ONDEMAND_SETUP ise
8: **end for**
9: onDemandSetUp *{defined below}*
10: **for all** incoming static edge $ise = C' \xrightarrow{static}$ THIS **do**
11: wait for CONFIRM_ONDEMAND_SETUP ise
12: **end for**
13: $mode \leftarrow$ VALID
14: **for all** in-sope out-going static edge ose **do**
15: CONFIRM_ONDEMAND_SETUP ose
16: **end for**
17:
18: **sub** onDemandSetUp

[10]To avoid setting up too many unnecessary dynamic edges, some special treatment for ongoing sub-transactions is needed. To this end here $p(T)$ only include those edges through which some sub-transactions of T have been initiated and these sub-transactions must have already ended. Those edges through which some sub-transactions have been initiated, but these sub-transactions have not ended yet, belong to the $s(T)$ declared below.

[11]When a dynamic reconfiguration request is received, the component(s) targeted for update is(are) known. Because the impact of the update is confined to the targeted component(s) and all components directly or indirectly depending on it(them), we only need to set up dynamic edges between these affected components. The set of these affected components can be figured out according to the static configuration, and passed along to the components by the on-demand set up requests. However, for readability the passing of the scope is omitted from the algorithm.

[12]This is to ensure that all in-scope components depended by THIS are in ONDEMAND or VALID mode and thus they response to dynamic dependence management requests properly.

19: **for all** $T \in localTxs()$ **do**
20: **let** $lfe = \text{THIS} \xrightarrow[root(T)]{future} \text{THIS}$, $lpe = \text{THIS} \xrightarrow[root(T)]{past} \text{THIS}$
21: $OES \leftarrow OES \cup \{lfe, lpe\}$, $IES \leftarrow IES \cup \{lfe, lpe\}$
22: **if** $root(T) = T$ **then**
23: **for all** $ose = \text{THIS} \xrightarrow{static} C \in f(T)$ **do**
24: **let** $fe = \text{THIS} \xrightarrow[T]{future} C$
25: **if** $fe \notin OES$ **then**
26: $OES \leftarrow OES \cup \{fe\}$
27: NOTIFY_FUTUE_ONDEMAND fe
28: **end if**
29: **end for**
30: **for all** $ose = \text{THIS} \xrightarrow{static} C \in p(T)$ **do**
31: **let** $pe = \text{THIS} \xrightarrow[T]{past} C$
32: **if** $pe \notin OES$ **then**
33: $OES \leftarrow OES \cup \{pe\}$
34: NOTIFY_PAST_ONDEMAND pe
35: **end if**
36: **end for**
37: **for all** $ose = \text{THIS} \xrightarrow{static} C \in s(T)$ **do** *{for ongoing sub-tx}*
38: **if** $ose \notin f(T)$ **then** *{do not create future edge from* THIS*, but require the host of the sub-tx to set up future edges}*
39: NOTIFY_SUB_FUTURE_ONDEMAND ose T
40: **end if**
41: **if** $ose \notin p(T)$ **then** *{do not create past edge from* THIS*now because the edge will be created at the end of the sub-tx. Require the host of the sub-tx to set up past edges}*
42: NOTIFY_SUB_PAST_ONDEMAND ose T
43: **end if**
44: **end for**
45: **end if**
46: **end for**
47: wait for ACKs for all above NOTIFY messages
48: **end sub**
49:
50: *Upon receiving* NOTIFY_FUTURE_ONDEMAND $xe = C \xrightarrow[T]{future} \text{THIS}$
51: $IES \leftarrow IES \cup \{xe\}$
52: **for all** $C' \in \{X \mid \text{THIS} \xrightarrow{static} X\}$ **do**
53: **let** $fe = \text{THIS} \xrightarrow[T]{future} C'$
54: **if** $fe \notin OES$ **then**
55: $OES \leftarrow OES \cup \{fe\}$
56: NOTIFY_FUTURE_ONDEMAND fe
57: wait for ACK_FUTURE_ONDEMAND fe
58: **end if**
59: **end for**
60: ACK_FUTURE_ONDEMAND xe
61:

62: *Upon receiving* NOTIFY_PAST_ONDEMAND $xe = C \xrightarrow[T]{past}$ THIS
63: $IES \leftarrow IES \cup \{xe\}$
64: **for all** $C' \in \{X \mid \text{THIS} \xrightarrow{static} X\}$ **do**
65: **let** $pe = \text{THIS} \xrightarrow[T]{past} C'$
66: **if** $pe \notin OES$ **then**
67: $OES \leftarrow OES \cup \{pe\}$
68: NOTIFY_PAST_ONDEMAND pe
69: wait for ACK_PAST_ONDEMAND pe
70: **end if**
71: **end for**
72: ACK_PAST_ONDEMAND xe
73:
74: *Upon receiving* NOTIFY_SUB_FUTURE_ONDEMAND $xse = C \xrightarrow{static}$ THIS T*:*
75: **let** $T' \leftarrow$ the sub-tx of T hosted by THIS component[13]
76: **for all** $ose = \text{THIS} \xrightarrow{static} C \in f(T')$ **do**
77: **let** $fe = \text{THIS} \xrightarrow[root(T')]{future} C$
78: **if** $fe \notin OES$ **then**
79: $OES \leftarrow OES \cup \{fe\}$
80: NOTIFY_FUTUE_ONDEMAND fe
81: **end if**
82: **end for**
83: **for all** $ose = \text{THIS} \xrightarrow{static} C \in s(T')$ **do** *{for ongoing sub-tx}*
84: **if** $ose \notin f(T')$ **then**
85: NOTIFY_SUB_FUTURE_ONDEMAND ose T'
86: **end if**
87: **end for**
88: wait for ACKs for all above NOTIFY messages
89: ACK_SUB_FUTURE_ONDEMAND xse
90:
91: *Upon receiving* NOTIFY_SUB_PAST_ONDEMAND $xse = C \xrightarrow{static}$ THIS T*:*
92: **let** $T' \leftarrow$ the sub-tx of T hosted by THIS component
93: **for all** $ose = \text{THIS} \xrightarrow{static} C \in p(T')$ **do**
94: **let** $pe = \text{THIS} \xrightarrow[root(T')]{past} C$
95: **if** $pe \notin OES$ **then**
96: $OES \leftarrow OES \cup \{pe\}$
97: NOTIFY_PAST_ONDEMAND pe
98: **end if**
99: **end for**
100: **for all** $ose = \text{THIS} \xrightarrow{static} C \in s(T')$ **do** *{for ongoing sub-tx}*
101: **if** $ose \notin p(T')$ **then**
102: NOTIFY_SUB_PAST_ONDEMAND ose T'
103: **end if**

[13]For simplicity T' is assumed to be unique. Details on possible cases that T' has not been initiated yet or has ended already are also omitted.

104: **end for**
105: wait for ACKs for all above NOTIFY messages
106: ACK_SUB_PAST_ONDEMAND xse