# Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates

Yves Vandewoude, Peter Ebraert,
Yolande Berbers, *Member*, *IEEE*, and Theo D'Hondt, *Member*, *IEEE*

**Abstract**—This paper revisits a problem that was identified by Kramer and Magee: placing a system in a consistent state before and after runtime changes. We show that their notion of quiescence as a necessary and sufficient condition for safe runtime changes is too strict and results in a significant disruption in the application being updated. In this paper, we introduce a weaker condition: tranquillity. We show that tranquillity is easier to obtain and less disruptive for the running application but still a sufficient condition to ensure application consistency. We present an implementation of our approach on a component middleware platform and experimentally verify the validity and practical applicability of our approach using data retrieved from a case study.

**Index Terms**—Distributed objects, componentware, language classifications, programming languages, software/software engineering, application-aware adaptation, support for adaptation, operating systems, software/software engineering, distributed objects, components, containers, language constructs and features.

## 1 Introduction

An intrinsic property of a successful software application is its need to evolve. In order to keep an existing application up to date, it must be continuously updated. Usually, the execution of such an update requires the application to be shut down and subsequently restarted.

This approach has a number of important disadvantages, which are not always acceptable. First, any state accumulated during the execution of the original version is lost when the application is shut down. Second, the application being updated is temporarily unavailable (disruption of service). At best, these two consequences are annoying: Data may need to be reentered or lengthy computations repeated. For many long-lasting or mission-critical applications, the shutting down of the application is unacceptable as it may result in loss of revenue or compromised safety. Finally, the shutting down of the running application strongly hinders self-adaptation. In research domains such as ubiquitous computing, applications are expected to adapt to an ever-changing environment. Such applications may wish to evolve at runtime by (re)loading parts of their functionality, depending on the context information they receive from their surroundings. A solution for this problem can be found in the domain of dynamic software evolution, in which a part of the application is updated while it remains active.

In order to guarantee application consistency during the application update, the system must first be placed in a consistent state before the runtime change is performed. After all, the new version of the application must be able to continue where the old application left off. This is not always the case if the application is updated at a random time. For example, if the implementation of a certain method has drastically changed (for example, a new algorithm is used), it is likely that no location exists in the new code from where the new version can complete the work done by the previous version. Simply waiting until active methods finish is insufficient too as the presence of transactions may result in additional consistency requirements.

The issue of when a piece of software is in the appropriate state for undergoing an update has been the focus of much research in the past. Most systems for dynamic updating either disregard the issue or put constraints on the systems that can be updated, (for example, they forbid the presence of such transactions). Of those systems that *do* address the issue, the work by Kramer and Magee [16]—who identified the quiescence criterion and proved that this criterion was sufficient to guarantee consistency during the update of a distributed system—was very influential. Their criterion, however, has the problem that it causes high disruption to the active program. In this paper, we propose an alternative criterion: tranquillity. We show that tranquillity drastically reduces disruption to the running program while remaining a sufficient condition for consistency. We show that, unlike quiescence, tranquillity is not proven to be reachable in bounded time. However, experiments show that, in

- *Y. Vandewoude and Y. Berbers are with the Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Heverlee, Belgium.*
*E-mail: {yves.vandewoude, yolande.berbers}@cs.kuleuven.be.*
- *P. Ebraert and T. D'Hondt are with the Programming Technology Lab, Vrije Universiteit Brussel, Plainlaan 2, B1050 Brussels, Belgium.*
*E-mail: {pebraert, tjdhondt}@vub.ac.be.*

*Manuscript received 22 Dec. 2006; revised 1 July 2007; accepted 23 July 2007; published online 8 Aug. 2007.*
*Recommended for acceptance by D. Binkley.*
*For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0294-1206.*
*Digital Object Identifier no. 10.1109/TSE.2007.70733.*

0098-5589/07/$25.00 © 2007 IEEE Published by the IEEE Computer Society

practice, tranquillity is nearly always reached within a short time frame.

The remainder of our paper is structured as follows: In Section 2, we present the quiescence criterion originally introduced by Kramer and Magee. We illustrate the highly disruptive behavior of this criterion and, in Section 3, we present our own alternative: tranquillity. We then show that tranquillity is less disruptive to the running application while, at the same time, remaining a sufficient condition for consistency in Section 4. Nothing comes for free and we give an analysis of the pros and cons of our criterion in Section 5. To illustrate the practical applicability, we describe a prototype implementation of our criterion on top of an existing component framework in Section 6 and further validate our approach with a case study in Section 7. We conclude this paper with related work and a summary of our findings.

## 2 The Concept of Quiescence

In the model by Kramer and Magee, a system is seen as a directed graph whose *nodes* are system entities and whose *arcs* are connections between those entities. Nodes can only affect each other's states via *transactions*, which consist of a sequence of messages that must be executed atomically (that is, either all messages are executed or none of them are). The node that starts the transaction is referred to as the *initiator* of the transaction. The model in [16] assumes that transactions complete in bounded time and that the initiator of a transaction is aware of its completion. Kramer and Magee abstract the status[1] of an application into a set of different configuration statuses for each node and consider two main statuses for each node, active and passive, whose definitions are given as follows:

**Definition 1 (active status).** *A node in the active status can initiate, accept, and service transactions.*

**Definition 2 (passive status).** *A node in the passive status must continue to accept and service transactions, but*

1. *it is not currently engaged in a transaction that it initiated and*
2. *it will not initiate new transactions.*

Kramer and Magee identify a passive status as a necessary but insufficient condition for updatability as a node may still be processing transactions that were initiated by other nodes. Therefore, they introduce a stronger concept:

**Definition 3 (quiescence).** *A node has a quiescent status 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, and*
4. *no transactions have been or will be initiated by other nodes that require service from this node.*

1. Kramer and Magee use the term *state* instead. In this paper, we choose to distinguish between the internal *state* of a node and the *status* that describes its condition in relation to the evolution process.

Although quiescence is a sufficient condition for updatability, it has the problem that enforcing quiescence often causes serious disruption to the running system. Not only must the node that is to be updated be put in a passive status, but this is also the case for every node that is directly or indirectly capable of initiating transactions on this node. This results in a serious drawback with respect to the impact the change has on the system [4]. We address this problem by introducing the concept of tranquillity.

## 3 The Concept of Tranquility

The tranquillity criterion is based on two observations:

1. There is no problem in replacing a node while a transaction is active as long as the node to be replaced is not involved in that transaction. This means that a node that has participated in an ongoing transaction may be replaced if it is certain that there will be no more future participation of that node in the transaction. It is equally permitted to replace a node that may, at some point in the future, participate in an ongoing transaction if it has not yet participated.
2. Using a black-box design for system nodes is a good approach for enhancing reusability and decoupling the system parts. This implies that the nodes may require services from other nodes they are connected to, but that they may never rely upon their implementation [19]. If all nodes are a black box by design, then all participants in a transaction either are the initiator of the transaction or are directly connected (adjacent) to the initiator. Nodes that are indirectly connected to the initiator can, by definition, not participate in a transaction driven by the initiator since their existence is unknown to the initiator. Note that any participant in the transaction can, in turn, initiate new transactions in response to a message they process. These *subtransactions*, however, are part of the implementation of this participant and are not known to the original initiator.

These two observations are exploited by the concept of *tranquillity*,[2] which we introduce as an appropriate status for updatability:

**Definition 4 (tranquillity).** *A node N is in a tranquil status 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, and*
4. *none of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.*

Quiescence is a stronger concept than tranquillity in the sense that quiescence implies tranquillity but not vice versa. Condition 3 of quiescence implies that the node is neither

2. As will be shown in Section 5, the tranquillity condition is not stable by itself. Hence, the name *tranquillity* may seem a little odd. In retrospect, the name *latency* better represents the semantics of our criterion, but it was decided to retain the current name for consistency with previous publications on the topic [21], [23].

actively processing a request nor waiting for a new request in an already active transaction. This trivially implies Condition 3 of tranquillity. Condition 4 of quiescence states that none of the adjacent nodes have initiated or will initiate a transaction in which $N$ participates. Hence, no such transaction is active, trivially implying Condition 4 of tranquillity. Tranquility does not imply quiescence, however, since it does not require that nodes connected with $N$ may not initiate new transactions that involve $N$. For tranquillity, nodes directly connected to $N$ must not reach a passive status.

Tranquility has the distinct advantage that it is much less disruptive than quiescence since only the affected node $N$ must be passivated. Although the third condition of tranquillity requires some adjacent nodes to finish a certain transaction, these nodes need not be completely passivated.

## 4 Tranquillity as a Sufficient Condition for Updatability

Although a weaker condition than quiescence, tranquillity is nevertheless a sufficient condition for updatability when two basic assumptions associated with an application are valid:

1. Like Kramer and Magee, we assume that both the original and the resulting configurations of the nodes are correct (that is, the update itself is sane). It is clear that the replacement of a `FileCompressor` node with an `ImageViewer` node is unlikely to result in a working system, no matter what update system is used to execute the update. Our work is only concerned with the correct execution of a certain update and not with trying to establish whether the update is correct in the first place.
2. Since each node should be reusable, it should only rely on external functionality if this functionality is declared to be public. This can only be achieved if the interactions between the nodes are made explicit and if no other dependencies between the nodes exist (for example, there may exist no dependencies between nodes that are not connected to one another).

The correctness of tranquillity under these assumptions is clear as a node in a tranquil status is, by definition, not executing code and can only be involved in an ongoing transaction if its participation in this transaction is 1) finished, 2) not yet begun, or 3) part of a subtransaction. In the first case, the update is clearly valid. In the second case, the validity of the update follows from the assumption of a valid resulting configuration: Transactions that have not yet begun may be executed by the new version (since the new application version contains the new version of the node). The correctness of the third case follows from a combination of the black-box principle and the correctness of the resulting configuration: Subtransactions are independent of the original transaction and may be executed by a different version than the original transaction. For clarity, we illustrate this principle with two examples.

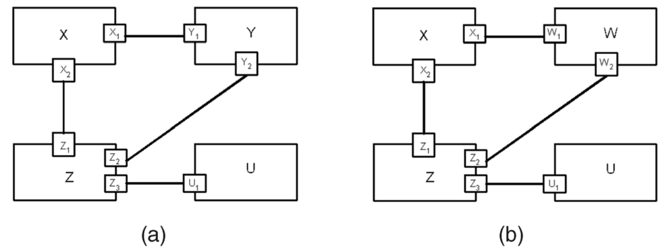



Fig. 1. An example of a component configuration before and after an update. (a) Original configuration. (b) Resulting configuration.

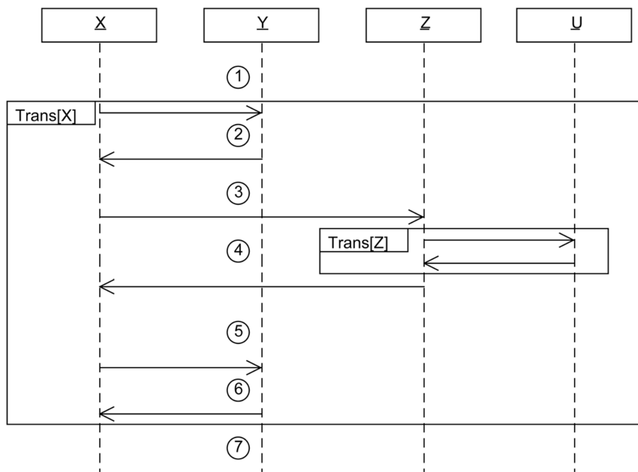
### 4.1 First Example

Consider the component-oriented system in Fig. 1, which consists of four components $(X, Y, Z, U)$ interconnected through their ports (denoted by a subscript, for example, $X_1$). Assume that one wishes to replace component $Y$ with a new version $W$. Furthermore, in the compositions shown in Fig. 1, component $X$ can execute a task for which it requires the assistance of its adjacent components ($Y$ and $Z$). The transaction that realizes the execution of this task is shown in Fig. 2. In Fig. 2a, the transaction is shown as it is executed in the current component configuration, whereas Fig. 2b shows the same transaction from the point of view of $X$'s implementation. Note that the subtransaction initiated by $Z$ is unknown to $X$.

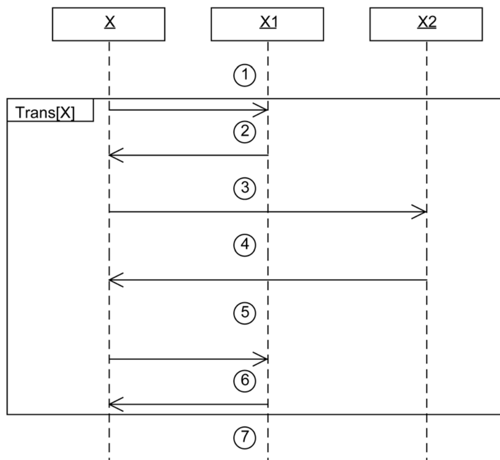
Assuming that there are no other transactions specified by either $X$ or $Z$ ($Y$'s adjacent components), the only times that $Y$ can be safely replaced are before and after the transaction (identified in Fig. 2 by the numbers 1 and 7). The periods identified by numbers 2 and 6 are characterized by execution in $Y$ itself and are therefore not suitable for replacement. Periods 3, 4, and 5 are entirely equivalent from $Y$'s perspective: $Y$ is an inactive and unknowing participant in a transaction initiated by $X$.

The replacement of $Y$ will not change the transaction itself since the transaction is entirely specified in $X$. Based on the validity of both the original and the resulting component configuration, this transaction will lead to a valid result with either $Y$ or $W$ ($Y$'s replacement) as the adjacent component connected to $X$. However, a valid result is not guaranteed if the transaction starts with the old version $Y$ and finishes with the new version $W$. This inconsistency occurs when $Y$ supports two symmetrical operations that are orthogonal to the working of $X$ but are nevertheless interrelated. For example, suppose $Y$ is a (de)compression component that offers two methods—`compress` and `decompress`—that return a (de)compressed version of the data supplied by the sender of the message. Component $X$ may wish to compress input data at the beginning of its task and decompress it again when it is done. Although it does not matter which compression algorithm is used, (indeed, the transaction is valid with both the old and the new version), correct functionality is not guaranteed if $Y$ is replaced by another component in the middle of the transaction. Note that this is the case whether or not $Y$ is a stateless or stateful component.

In this transaction, both the condition of quiescence and the condition of tranquillity forbid replacement at times 2 to 6. However, the tranquillity condition allows the replacement of $Y$ at the beginning or the end of the transaction (times 1 and 7). This is not the case for quiescence as

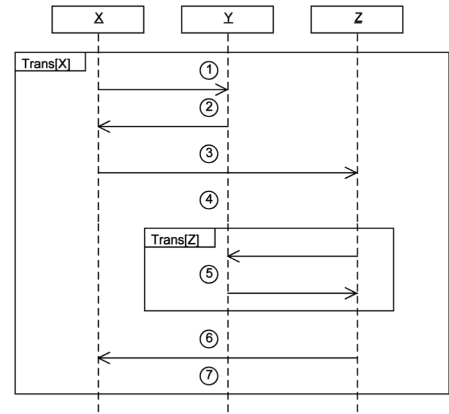



(a)

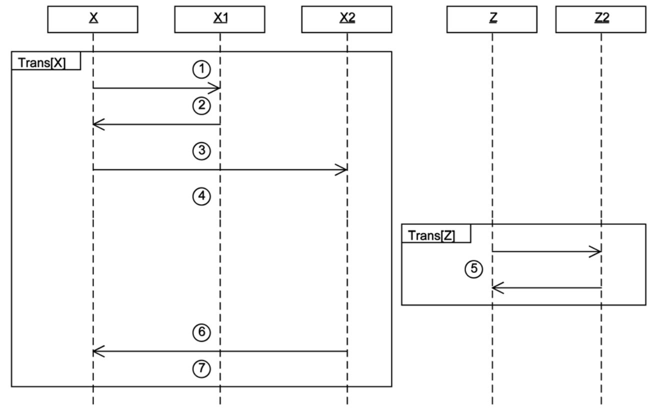



(b)

Fig. 2. A transaction in which $Y$ participates prevents $Y$ from being updated. (a) Transaction as executed. (b) Transaction as perceived by $X$.




(a)




(b)

Fig. 3. $Y$ is used in a transaction from $X$ and in a subtransaction initiated by $Z$. (a) Transaction as executed. (b) Transaction as perceived by $X$ and $Z$.

quiescence also requires that *no transactions have been or will be initiated by other nodes that require service from this node*, which requires $X$ (and $Z$ for that matter) to be completely passivated. Exploiting the validity of the resulting composition, the tranquillity requirement allows much quicker replacement while still ensuring consistency.

### 4.2 Second Example

A slightly more complex scenario is shown in Fig. 3. This scenario assumes the same initial and resulting component configuration but assumes a different active transaction. At time 4, $Y$ may be replaced according to our tranquillity definition since $Y$ is currently not involved in a transaction that it initiated, the transaction by $Z$ has not started yet and further execution of the transaction by $X$ no longer directly involves $Y$ from $X$'s point of view. As it turns out, it is indeed correct that $Y$ can be replaced at this point. This is shown in Fig. 3b. The transaction initiated by $Z$ is independent of the transaction initiated by $X$. Since both the initial and the resulting configurations are correct, the transaction of $Z$ leads to correct results using either $Y$ or $W$ as its participant. The ongoing transaction initiated by $X$ is unaware of the transaction initiated by $Z$. Due to their independence, it is perfectly possible that the transaction shown in Fig. 3 starts with $Y$ and finishes with the new version $W$. A replacement of $Y$ at time 4 is not permitted using quiescence as a condition since quiescence does not take into account the independence of the two transactions.

## 5 Reachability of the Tranquillity Condition

Although tranquillity is a sufficient condition for application consistency during a dynamic reconfiguration, there are disadvantages to our criterion as well. The most important drawback is that it is not guaranteed that a tranquil status will ever be reached for a given component. This is the case when this component is used in an infinite sequence of interleaving transactions. An example of such a case is shown in Fig. 4. The figure shows two interleaving transactions that are infinitely repeated (only the first and the beginning of the second iteration are shown). Because $Y$ is always active in a transaction in which it still needs to participate, it can never reach tranquillity without directing $X$ and $Z$ to a passive status, (which would imply quiescence).

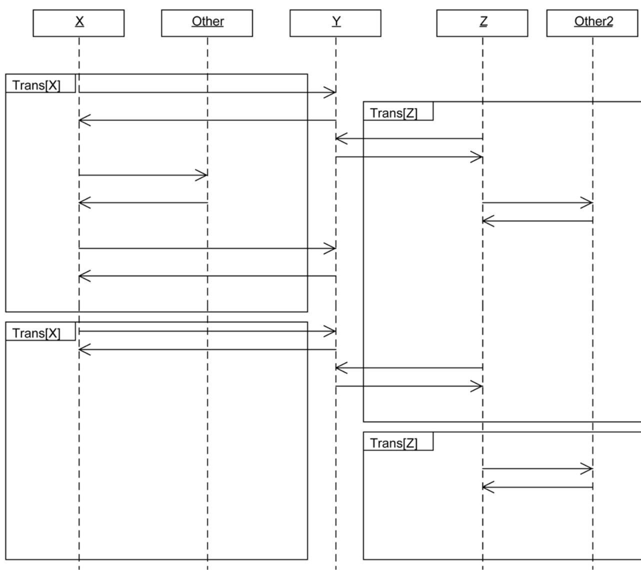



Fig. 4. A scenario in which component $Y$ will never reach tranquillity.

A second disadvantage of tranquillity is that the criterion is not stable by itself. As soon as a node $N$ is in a tranquil state, all interactions between that node and its environment must be blocked in order to guarantee that the node remains in a tranquil state. This is not the case with quiescence, where the passivation of all nodes that can directly or indirectly initiate a new transaction on $N$ ensures that $N$ remains in a quiescent state until the passivated nodes are explicitly reactivated.

It may seem contradictory that tranquillity is a weaker condition than quiescence but that it cannot be reached in bounded time, whereas quiescence can. This is because tranquillity can occur naturally, whereas a system must be actively driven in a quiescent status. Clearly, tranquillity *is* reachable in bounded time by directing the system to a quiescent status, but that nullifies the advantages of the tranquillity criterion.

Nevertheless, because tranquillity does not always occur in bounded time, any system that implements dynamic updates using the tranquillity condition must implement a fallback mechanism to quiescence for when tranquillity is never reached. It should be noted that these situations are rather rare in practice [10, pp. 428-429] and that, in most cases, the tranquillity condition occurs within a short period of time.

# 6 Implementation on Component Middleware

A prototype implementation was developed as an extension to a general-purpose component middleware platform: DRACO. The implementation allows the middleware to put active components in a tranquil status upon demand. To do so, the system simply observes messages between the components until tranquillity is observed, after which it blocks all incoming and outgoing messages by that component. When the tranquillity status cannot be reached, it transparently falls back to the quiescence requirement. Although mere observation is theoretically sufficient, the following sections will show that our prototype implementation does block messages when waiting for tranquillity. This is mainly for practical reasons, such as ensuring that the middleware has sufficient time to check all conditions and to ease the isolation of the component as soon as tranquillity is reached.

We begin this section with the introduction of the main concepts supported by the DRACO methodology. A full description of the component model, the language, or its tool chain is outside the scope of this paper and we restrict ourselves to the core concepts of the methodology and how these concepts map to the model by Kramer and Magee. Relevant implementation aspects of the component middleware environment are discussed in Section 6.2. Finally, we present a detailed description of how the Live Update Extension Module (LUM) realizes updatability using the tranquillity condition.

## 6.1 The DRACO Component Methodology

In DRACO, *components* are units of functionality that are implemented as a highly cohesive group of Java classes. Once instantiated, they represent a tightly coupled group of objects. Interconnection between components is achieved by means of *connectors.* According to Aldrich et al. [1], a connector is a reusable design element that supports a particular style of component interactions. DRACO assumes the interaction style that was defined in the SEESCOA project [2], [20]. In this model, components communicate by asynchronously sending messages through external interfaces that are formally specified using *ports.* Connectors attach to these ports and implement a pipe-like construct, which makes relaying or intercepting communication easy to achieve. The conditions of explicit communication that were assumed in Section 4 are therefore clearly met in the DRACO component model.

In order to map our component model onto the model assumed by Kramer and Magee, it suffices to consider our components to be the nodes and our connectors to be the arcs of their directed graph. The bidirectional nature of connectors can easily be modeled using two directed arcs with opposite directions. Furthermore, in the DRACO component model, the state of components can only be changed by message interaction with other components. Finally, the DRACO component model ensures that all message sequences complete in bounded time.

## 6.2 An Extensible Middleware Platform

The DRACO middleware platform was designed with extensibility in mind and offers an extensive API that can be used by extension modules to change the behavior of the core system. Its architecture consists of five core modules:

1. the component manager, responsible for loading and instantiating component instances,
2. the message manager, responsible for the message delivery process,
3. the scheduler, responsible for scheduling messages that have been sent and that are awaiting execution,
4. the connector manager, responsible for (dis)connecting ports, and
5. the module manager, responsible for adding extension modules to the core system at runtime.

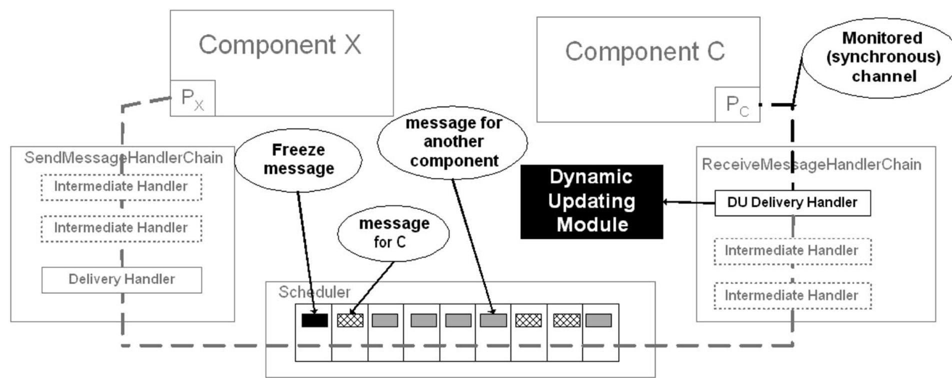



Fig. 5. Situation after initiation of the update of $C$.

Message delivery is achieved in three stages. The first stage is the transmission of the message by the originating component. In this stage, the message passes through a sequence of message handlers (who can transform messages that pass through it) until it is handed over to the scheduler. This first stage is executed by the thread currently active in the originating component. Because interaction with the scheduler is nonblocking, message sending is asynchronous. In the second stage, the message awaits its execution inside a message queue from the scheduler. Finally, in the third stage, the scheduler's *worker threads* continuously fetch messages, pass these messages through a similar handler chain, and, finally, deliver them to the receiving component. The scheduler guarantees that the order of messages over a given connector is preserved and that messages are delivered sequentially.

The message handler mechanism opens up the delivery process as extension modules can insert or remove custom handlers that change the default behavior. DRACO also makes extensive use of the observer pattern and allows for extension modules to subscribe themselves to a large number of events that are triggered before and after all important actions such as component (un)loading, (dis)-connecting, and message sending.

### 6.3 Live Update Extension Module

LUM is an extension of the core DRACO system that allows components to be replaced by a new version at runtime. After the application maintainer has specified that a certain component $C$ needs to be replaced, LUM places that component in a tranquil status. The module then transfers the state contained in the old version to the new version, rewires the connectors, and activates the new version. This paper is only concerned with the first step. The following sections describe how the tranquil status is reached and how the module falls back to quiescence if tranquillity is not attainable.

#### 6.3.1 Enforcing Passivity

Since tranquillity encompasses all requirements of passivity, LUM will first direct $C$ to a passive status before it enforces the other tranquillity conditions. This passive status is attained by ensuring that

1. the component is not actively executing a message and
2. the inflow of new messages to the component is restricted.

If no messages are executing, the first passivity requirement is trivially fulfilled. In addition, no new transactions can be initiated by $C$ because messages can only be sent out by a component as part of code execution that in itself is triggered by a message.

LUM achieves passivity by restricting all incoming traffic to $C$. It does so by replacing the standard delivery message handler on the receiving message chain of each port of $C$ by a custom delivery message handler (see Fig. 5). Although Fig. 5 only shows one connected port on $C$, the situation is analogous for all other connected ports. To guarantee that all interactions with $C$ are controlled, LUM registers itself with the connector manager to temporarily prevent changes to the connections of $C$'s ports. LUM then sends a `Freeze` message to a randomly selected connected port of $C$.

As illustrated in Fig. 5, any number of messages with component $C$ as their destination can be present in the scheduler queue at the time the `Freeze` message is sent out by LUM. Since the custom delivery message handlers are only inserted in the chains associated with ports of component $C$, messages that are intended for other components are unaffected by the replacements in the delivery message chains, which reduces unnecessary overhead.

The custom handler introduced by LUM initially mimics the behavior of the original delivery handler: It executes the method associated with the message on the component and then terminates (thus returning control through the delivery chain all the way back to the scheduler). This behavior changes after it encounters the `Freeze` message. At that time, the message is executed if it is supported by the component (allowing the designer of the component to specify the custom cleanup code) or ignored otherwise. Afterward, however, control is no longer returned to the scheduler. In addition, other custom delivery handlers associated with a port of $C$ halt before executing the message, effectively terminating all communication with $C$. Although no new messages can reach $C$, the passivity requirements have not been fulfilled so far.

First, the component may still be executing the code in a dedicated thread. For the sake of brevity, we are not including a full description of how such threads are handled by the component system. For this paper, it suffices to say that the scheduler of DRACO is aware of such threads and that it can safely preempt the majority of such threads without leaving the component in an inconsistent state. Whether or not DRACO was able to preempt the thread, LUM delays the update of the component until its execution has terminated. We refer the reader to [22] for more details.

Second, a transaction initiated by $C$ can still be active. For example, at time 3 in Fig. 2a, component $X$ is inactive and awaiting a response from $Z$. This leads us to the problem that LUM must be able to determine when a component is actively engaged in a transaction it initiated. When a transaction consists of asynchronous messages (which is the case in DRACO), this cannot be determined automatically unless the component that drives the transaction provides this information. Our implementation assumes that each component implements a method that returns (a Boolean stating) whether or not the component is active in a transaction it initiated. LUM queries this information and resumes message delivery on a message-per-message basis as long as $C$ is in such a transaction. After each message, the expression embedded in $C$ is reevaluated until the component has terminated its transaction. At this moment, $C$ has reached passivity and will maintain this status since no further messages are allowed into $C$.

### 6.3.2 Ensuring the Additional Tranquillity Constraints

Before the passivated component $C$ may be replaced, the additional conditions of tranquillity must be met. LUM does this by querying all adjacent components of $C$ and determining whether these components are involved in a transaction they initiated. If so, LUM requests from each of these components a list of all ports that have participated in their transaction and a list of those ports that are still required to finish the transaction.[3] If a port of $C$ is attached to a port present in both lists, the requirements of tranquillity are not fulfilled.

LUM then starts to monitor all messages entering the adjacent components of $C$ by replacing their message delivery handlers. Whenever a message is delivered to these components, the transaction requirement is reevaluated. As the completion of the transaction requires the participation of $C$, it is necessary that $C$ accept messages involving the transaction. The custom delivery message handlers associated with the ports of $C$ will therefore resume message delivery, again on a message-per-message basis. After each message delivery, both to $C$ and to its adjacent components, the conditions are rechecked. In the majority of cases, neither complex systems of interleaved transactions nor transactions with circular dependencies are an issue [10, pp. 428-429] and a tranquil point is reached for $C$ in a relatively short period of time. Once this tranquil point is reached, all messages to $C$ are prevented and the tranquillity condition is preserved for the duration of the update.

3. Although it may seem a large overhead to generate this data, the information can be automatically generated from a state machine that describes the transaction. This state machine can be automatically derived from message sequence charts, such as those used in the figures of this paper.

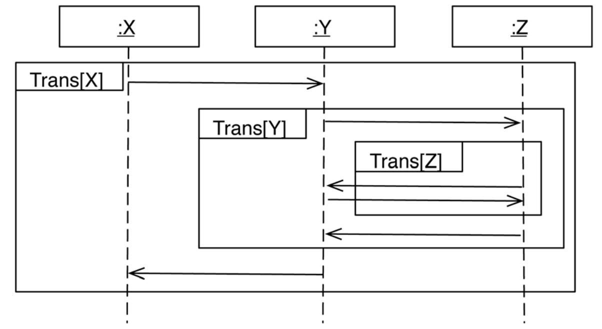



Fig. 6. A system with cyclic dependencies.

### 6.3.3 Fallback to Quiescence

Because reaching tranquillity in bounded time cannot be ensured in general (Section 5), LUM also keeps an internal timer. If, after a predetermined time frame, tranquillity has not been reached for $C$, then the system falls back to the more stringent and invasive requirement of quiescence, which was proven to be reachable by Kramer and Magee in [16]. Their model assumes that a node has knowledge of whether its actions are part of a transaction initiated by another node. As this assumption is not valid in our component model, ensuring the reachability of quiescence in bounded time needs to be examined further.

The problem is caused by dependent transactions, which Kramer and Magee define as follows:

**Definition 5 (dependent transaction).** *A dependent transaction is a two-party transaction whose completion may depend on the completion of other consequent transactions.*

In other words, $t_i$ is a dependent transaction if there exists a chain of transactions $t_i, t_j, \ldots, t_s$ in which each, with the exception of $t_s$, may depend for completion on the completion of its (consequent) successor transaction. Dependent transactions and their potential consequent(s) are denoted as dependent/consequent(s). Cycles are not forbidden, but the model by Kramer and Magee does assume that the transactions still complete in bounded time and that deadlocks are avoided. It is also required that the initiator of a dependent transaction be informed of the completion of consequent transactions because, otherwise, a component cannot determine when the transactions it has initiated have been completed and, hence, when it has reached passive status. Each of these assumptions is reasonable and also valid in our own component model.

The problem with dependent transactions is that the passive status may not be reachable for components utilizing dependent transactions. Assume three components, $X$, $Y$, and $Z$, as depicted in Fig. 6. Suppose that component $Z$ is in a passive state and $X$ has initiated transaction $\mathrm{TRANS}(X)$. In this situation, transaction $\mathrm{TRANS}(X)$ cannot be completed because $\mathrm{TRANS}(Y)$ cannot be completed because $\mathrm{TRANS}(Z)$ may not be initiated by $Z$ as it is in a passive state.

Consequently, neither $X$ nor $Y$ can move into the passive state in bounded time.

The solution proposed by Kramer and Magee is to generalize the definition of passive status to include the means for dependent transactions to be completed:

**Definition 6 (Generalized Passive Status).** *A component in the generalized passive status must accept and service transactions and* initiate consequent transactions, *but*

1. *it is not currently engaged in a (nonconsequent) transaction that it initiated and*
2. *it will not initiate new (nonconsequent) transactions.*

To implement this solution, we have to cope with the fact that DRACO components are not aware of their participation in transactions they did not initiate. It is therefore not possible for a component to identify those transactions that are consequent and those that are not. Adding this information to the component code is unacceptable, however, as this would increase implicit coupling between components and strongly hinder reuse. In DRACO, this problem is solved by performing additional bookkeeping when messages are sent or received [17]. Whenever a component sends out messages in the context of a transaction it initiates, it tags these messages. The DRACO message delivery system recognizes these tags and transparently forwards the tag to all messages sent out as part of that transaction.

As a first step toward placing a component $C$ in a quiescent state, LUM conveniently makes use of this feature when it composes a set of ongoing transactions that must finish before quiescence can be reached. To do so, LUM queries all of the adjacent components of $C$, checks which ongoing transactions involve $C$ as a participant, and stores them in the `Initial List`.

**Algorithm 1** Receive(m)

**if** tag($m$) part of InitialList **then**
  $messageThread \leftarrow$ current thread
  $outId \leftarrow 0$
  $struct \leftarrow < messageThread, C, \text{tag}(m), outId >$
  Execute($m$)
  Rebuild InitialList
  **if** InitialList is empty **then**
    Quiescence Reached
  **end if**
**else** //*Not part of an ongoing transaction*
  Queue($m$) at the deliveryMessageHandler
**end if**

Whenever a message is received by $C$ or by one of its adjacent components, their message handler intercepts the message and checks whether the message is a part of a transaction of the `Initial List`. If so, the message handler stores a tuple containing the ID of the current thread, the ID of the current component, the tag of the received message, and a new tag for the outgoing message. Afterward, it executes the message and checks whether the `Initial List` is empty, which would mean quiescence has been reached for $C$. If the message is not part of a transaction in the `Initial List`, it is queued as its execution is not required for reaching quiescence for $C$.

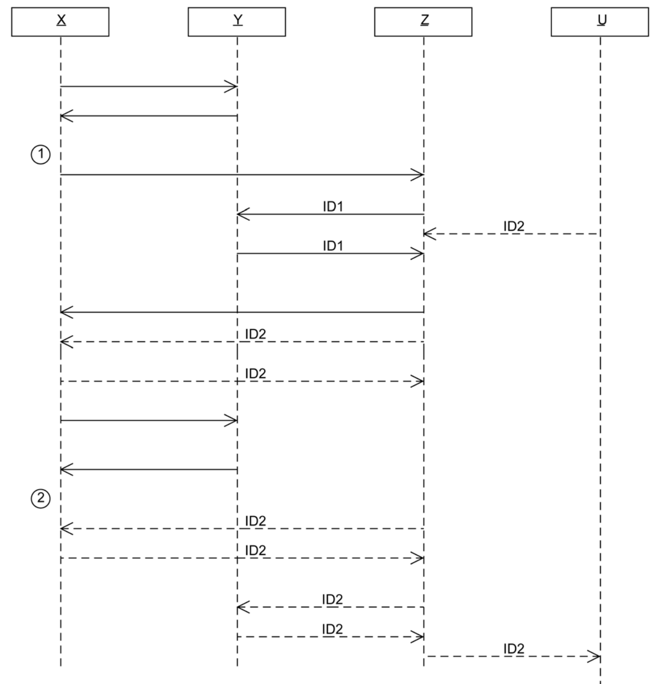



Fig. 7. Two-phase optimization for achieving quiescence.

**Algorithm 2** Send(m)

$sendThread \leftarrow$ current thread
$struct \leftarrow$ LUM.get($sendThread$)
**if** exists($struct$) **then**
  S.outId++
  tag($m$) $\leftarrow S.tag + "." + S.N + S.outId$
**end if**
messageHandler.deliver($m$)

Whenever a message is sent by $C$ or one of its adjacent components, the message handler of the sending component intercepts this message, looks for the tuple that corresponds to the current thread ID, and uses this information as a basis for a new tag that correctly identifies the message as part of the transaction.

As soon as all transactions in the initial list have ended, only tagged messages are delivered to the component. Since all transactions are assumed to be bound in time, this moment is certain to occur in bounded time. In the example in Fig. 7, this moment is identified by label 2. Because all other messages are queued by the delivery message handlers, ongoing transactions can be completed, but no new transactions can be initiated.

The above implementation assumes that all messages belonging to a transaction are tagged so that, when the `InitialList` is constructed, the tags of active transactions can be retrieved. It is possible to relax this requirement in order to minimize overhead during normal application execution. If the components are still aware of the transactions they have initiated but the messages that belong to these transactions and the resulting dependent transactions are not tagged, then LUM can still reach quiescence, albeit using a slightly more complex algorithm consisting of two phases (Fig. 7). In the first phase, all messages are delivered to their destination because it is not

Fig. 8. A trivial scenario in which the atomic replacement of components could be required.

known to which transaction each message belongs. In addition, LUM starts tagging all newly initiated transactions that involve $C$. The second phase starts as soon as the `InitialList` is empty. At this moment, identified by label 2, we have the situation assumed in the basic algorithm described above since all ongoing transactions involving $C$ are now tagged.

### 6.4 Atomic Deactivation of Multiple Components

Although our methodology tries to keep dependencies between different component instances to a minimum, there are cases where multiple components must be replaced at the same time. One very common example of where this functionality is necessary is in the presence of so-called glue components: a component that was specifically designed to bridge semantic and/or syntactic gaps between other components in the composition. Although the DRACO middleware platform does offer syntactic bridging with flexible connectors, the presence of glue components is a side effect of our methodology and cannot be avoided in complex component scenarios. We consider glue components to be the lesser of two evils: The only way to avoid a glue code is to design different components so that their interfaces are compatible. This strongly increases implicit coupling between the components and makes them less reusable.

Consider the example in Fig. 8 in which a camera component sends out images in a certain format. In order to show the feed from this camera using an independently developed image viewer component, a conversion component is likely to be required. It is easy to consider a scenario in which the image viewer is replaced by an incompatible version for which another conversion component would be required.

#### *6.4.1 The Problem: Messages in Transit*

Scenarios of atomic replacement of multiple components complicate the deactivation process. Both the tranquillity and the quiescence conditions remain valid. However, there is one important difference from the theory described in the previous sections: the necessity of taking into account messages in transit. Assume the component composition in Fig. 9, which is similar to the evolution scenario in Fig. 1 but with the additional requirement that *both* $X$ and $Y$ are to be replaced atomically.

If both components $X$ and $Y$ are to be replaced atomically, one must make sure that no messages of $X$ with destination $Y$ are in transit at the moment when the replacement takes place. After all, there is no guarantee that the component composition will behave correctly if messages from $X$ are received by $W$ (or messages from $Y$ by $V$, for that matter). Note that this was not a problem in the previous section due to the correctness of the resulting configuration. If only $Y$ is replaced by $W$, it follows from the correctness of the resulting composition (which contains both $X$ and $W$) that the reception of a message from $X$ by $W$ is acceptable from the composition designer's perspective. In the new example depicted in Fig. 9, however, neither the original nor the resulting configuration contains a combination of either $X$ and $W$ or $Y$ and $V$. It is clear that the dynamic update system must therefore prevent messages being sent out by $X$ from ever reaching $W$ or vice versa.

Fortunately, the criteria of tranquillity and quiescence are sufficient to detect such problems in the presence of transactions. Consider Fig. 10, which was modified from Fig. 2b to reflect the fact that message delivery is not instant.

At the time indicated on the figure, a message is in transit from component $X$ to component $Z$. This causes a problem if, at this moment, both $X$ and $Z$ are replaced atomically. However, neither $X$ nor $Z$ can be replaced at this moment. $X$ is active in a transaction it initiated and $Z$ fails the last requirement of tranquillity since $X$ (an adjacent component of $Z$) is engaged in a transaction in which $Z$ had already participated and might participate in the future. Note that, from $X$'s perspective, $Z$ was already active in the transaction as soon as the message for $Z$ was sent. In general, the requirements in Sections 2 and 3 will never permit the replacement of both parties if a message

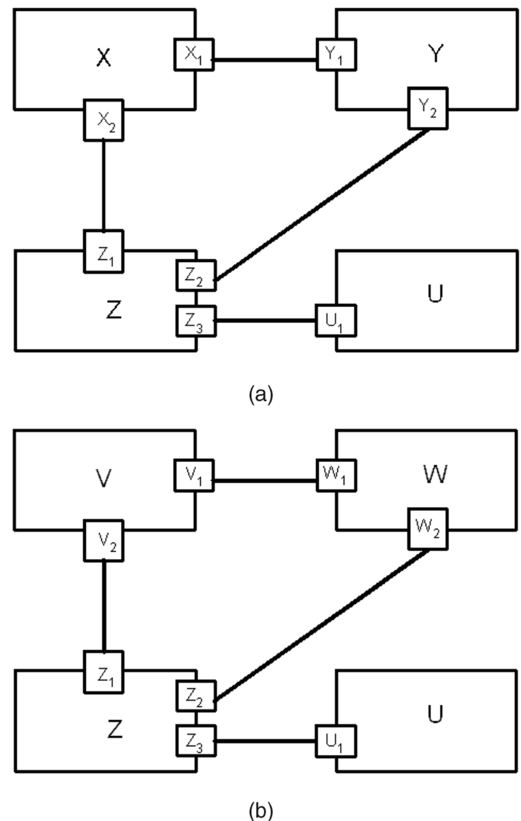



Fig. 9. The two components $X$ and $Y$ are atomically replaced by $V$ and $U$, respectively. (a) Original configuration. (b) Resulting configuration.

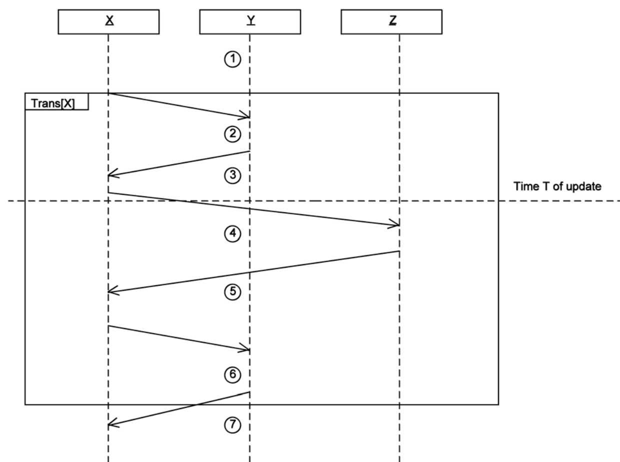



Fig. 10. The tranquillity and quiescence requirements are sufficient to deal with messages in transit in the context of transactions.

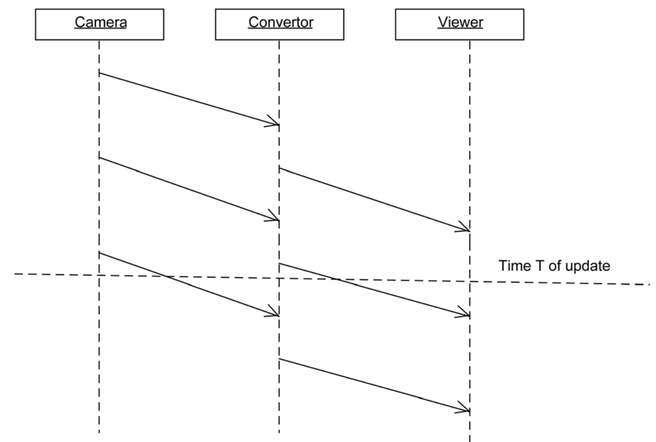



Fig. 11. At time $T$, a message sent from the convertor to the viewer is in transit. Each component implements a simple pipe-and-filter architecture and no transactions are active in the system. The update should not be allowed to proceed without first halting the message stream between the convertor and the viewer.

belonging to a transaction is in transit since, at the very least, the sender is by definition involved in a transaction it initiated.

Problems arise, however, when no transactions are involved. Fig. 11 revisits the Camera-Viewer scenario introduced earlier in this section and shows three components: the camera, the convertor, and the viewer. At the indicated time, a request is made to replace both the convertor and the viewer atomically. At the indicated time, however, two messages are in transit. The first message has been sent by the camera but has not yet reached the convertor. This message is not an issue because both the old and the new convertors are capable of dealing with the message due to the correctness of the resulting configuration. The second message in transit is logically located in the connector between the convertor and the viewer. This message *is* an issue since the viewer component of the new configuration may not be equipped to deal with the message in transit.

#### 6.4.2 A Solution: Virtual Transactions

Two strategies can be used to address messages in transit:

1. *Corrective.* The corrective approach carries out the replacement while disregarding potential messages in transit. After replacement, measurements are taken to ensure that the messages in transit are found and transformed before any of them are delivered to the new version of the receiving component. This technique has the major drawback that it requires significant application domain knowledge to implement the message conversion. The approach also adds to the complexity and, hence, the time required to carry out the update and, therefore, increases disruption in the system.
2. *Preventive.* The preventive approach ensures that there are no such messages to begin with. The approach relies on the update algorithm to ensure that the message sender (in our example, the conversion component) terminates its transmissions and that all messages in transit have been delivered before the components are updated.

In our work, we have opted for the second choice because of its fully automatic and application-independent characteristics. In addition, its implementation allows the reuse of all algorithms and techniques discussed for single-component deactivation. The solution is based on the observation that the problem with messages in transit only occurs in the absence of transactions. Therefore, if every message were part of a transaction, the problem would not occur. This is exactly the approach taken by LUM: virtual transactions.

LUM considers each message itself as a transaction that starts at the moment when it is sent (enters the connector) and ends at the moment when it arrives at its destination (leaves the connector). Note that the components themselves are unaware of these *virtual* transactions. When LUM needs to atomically replace multiple components, it just uses the same algorithms as in the case where only one component needs to be deactivated. When applied to the camera example in Fig. 11, the update will not be able to proceed since the `Convertor` component is still active in a transaction it has initiated.

## 7 Experimental Validation

In order to verify the practical applicability of tranquillity, we have implemented a component-based Web shop application. The application provides the core functionalities for selling articles over the Web. It consists of five components: a `User Controller`, a `Currency Converter`, a `Price Calculator`, a `Product Localizer`, and an `Interaction Module`.

### 7.1 Architecture

Fig. 12 shows the architecture of the Web shop application. The `User Controller` coordinates the enquiries and the purchases of the clients. The `Currency Converter` maintains the conversion rates of the different currencies and performs conversions whenever required. The `Price Calculator` contains the business logics of the Web shop. It is capable of calculating the total price of a client's

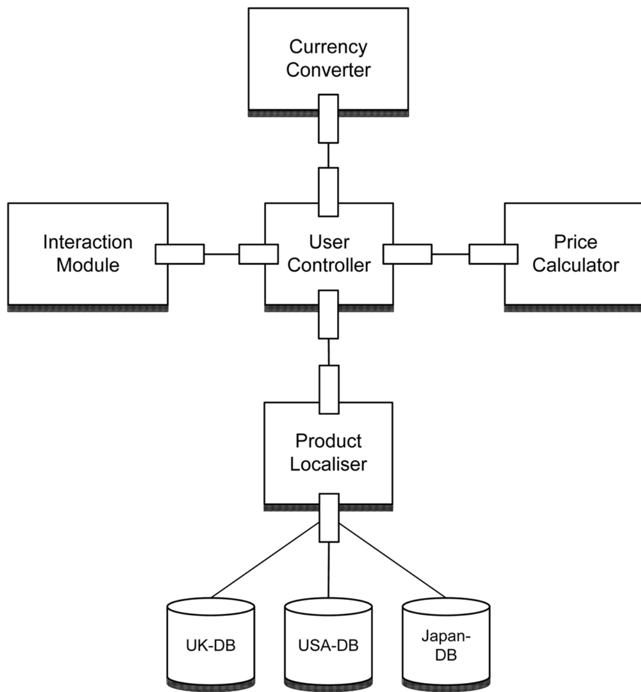



Fig. 12. The component-based Web shop.

shopping basket, taking into account the current promotions and delivery costs. The `Product Localizer` maintains links to local stock databases. It is responsible for filtering the items, depending on the location of the user. The `Interaction Module` is responsible for interacting with the users. It provides the graphical user interfaces for querying and for purchasing products. An important property of this module is that it always displays the prices in the user's preferred currency.

For the sake of brevity, we do not explain the detailed mechanics of a purchasing scenario. Instead, we focus on the price conversions that occur in a typical purchasing scenario (illustrated in Fig. 13). Whenever the user executes a product query, the `Interaction Module` shows all products in the user's preferred currency. The `Price Calculator`, however, expects all items to be priced in US dollars to accurately compute discounts and shipping. It is of the utmost importance that the same conversion rate be used throughout the entire transaction.

### 7.2 Runtime Adaptation Experiment

In our experiment, we update the `CurrencyConvertor` component with new rates. Note that it does not matter whether this change is anticipated (that is, the component provides functionality to update its state) or requires a component replacement. In both cases, the `Currency-Convertor` must be in a tranquil state in order to preserve application consistency. We implemented and executed this change scenario on the DRACO component middleware platform introduced earlier and recorded how long it took to reach a tranquil state during the replacement of the `CurrencyConvertor`. The update of the `CurrencyCon-vertor` was executed under different system loads: 10, 50, and 100 concurrent clients, respectively, each emitting one request per second with random pauses of 100 ms in order to ensure a more realistic and homogeneous system load. Each of these three scenarios was measured 10 times and both the mean and the standard deviation of the measured durations are shown in Table 1. The duration is measured from the moment the update is requested by the user until tranquillity occurs. It is expressed both in absolute time (seconds) and in relative terms with respect to the number of messages inspected before tranquillity is attained (*number of ticks*). The data in Table 1 clearly illustrate that the time required to reach a tranquil status varies significantly between different experiments, even for identical setups. This is because tranquillity is heavily influenced by the order in which requests from different clients arrive and whether the transactions of the clients heavily overlap. Despite this high degree of variability, the theoretical scenario where tranquillity is never reached does not often present itself in reality. Actually, in none of the 40 executions of our evolution scenario did we require a fallback to quiescence to accomplish the update. Another important result from our tests, although not shown in Table 1, is that, in every test case, a consistent application condition was reached after the update.

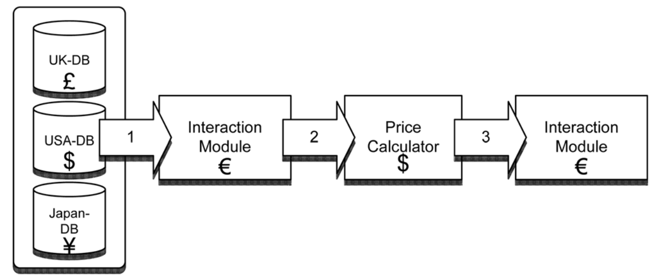



Fig. 13. The different currency conversions.

If the distribution of the data in Table 1 is known, the data can be used to calculate additional information, such as a 95 percent percentile upper bound (a time frame in which tranquillity will be reached 95 percent of the time) or the probability that tranquillity will not be reached within a user-specified time frame. Since the statistical distribution of the data in Table 1 cannot be derived from 10 experiments, we have executed an additional 200 experiments under the light system load (10 concurrent clients). The histogram that summarizes the results from these additional experiments is shown in Fig. 14. The assumption of a

TABLE 1
Experimentally Measured Durations to Reach a Tranquil State

| | 5 Clients | 50 Clients | 100 Clients |
|---|---|---|---|
| *Time (in seconds)* | $40.6 \pm 31.2$ | $120.6 \pm 67.9$ | $197.2 \pm 49.3$ |
| *Duration (in ticks)* | $219.7 \pm 145.1$ | $1124 \pm 509.5$ | $3494.3 \pm 260.5$ |

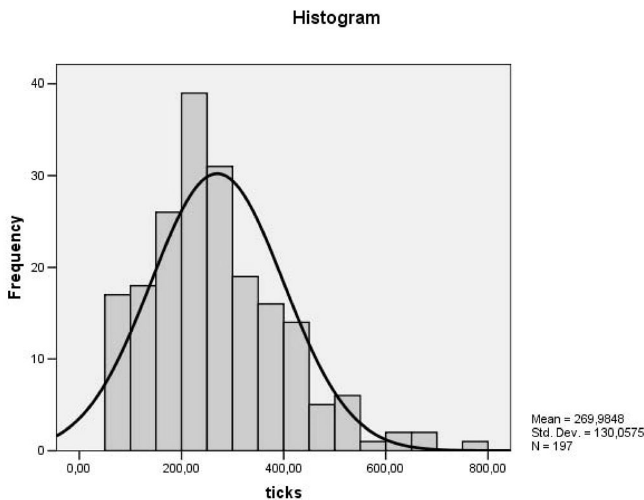



Fig. 14. This histogram shows 197 additional experiments with 10 concurrent clients. The horizontal axis shows the number of ticks until the tranquil status is reached.

normal distribution was not contradicted by the One-Sample Kolmogorov-Smirnov Test for normality. Note that the mean value and standard deviation roughly correspond to the data in Table 1, which indicates that the 10 original experiments are reasonable approximations for the higher system loads as well.

## 8 Related Work

Although we are weakening the notion of quiescence, there are others who have implemented the notion of quiescence as presented in [16]. The work in [5] shows an implementation of the quiescence model for distributed component systems. Gerlach and Baelen [8] present resource-aware components. These are components that are aware of their environment and can react to changes in it. Taking this further brings us to the world of ambient software, where components adapt their behavior using information from the outer world [24]. These approaches, however, consistently break the black-box design principle, thus lowering reusability.

The notion of tranquillity is not only applicable in component systems [18], [19] but also in all other paradigms that allow modularization and explicit interaction. We have found applications of the notion of quiescence in procedural programming, in service-oriented programming, and in object-oriented systems. In all of these approaches, the notion of tranquillity could be introduced for enhancing the dynamic updatability.

As early as 1976, Fabry presented a system allowing for dynamic changes of abstract data types written in procedural languages [6]. Other systems in this area were developed by Gupta [11], Hicks [13], and Hofmeister. The latter even stated in [14] that the notion of quiescence is too strong and that entities can be safe without enforcing quiescence. Her approach still does not offer support for black-box entities, however.

Service-oriented systems [3] are decomposed into different entities that provide and request services. Two entities are connected by service contracts [7]. This explicit linking of entities enhances the decoupling. This explains why Ketfi and Belkhatir [15] introduce the notion of quiescence for service-oriented systems.

Our approach is also applicable to ordinary object-oriented systems. The Dynamically Alterable System (DAS) [9] is an operating system from the late 1970s. It supports the replacement of an object by another one with the same interface. In DAS, the in and out-operations on objects are first class, ensuring less coupling between the objects and allowing data restructuring. In [12], Gupta et al. show how object-oriented systems should be updated dynamically. They also claim that the programs should be in a quiescent state before the updates can be carried out.

## 9 Conclusion

This paper addresses the problem of state consistency before and after a dynamic change. The problem was originally identified by Kramer and Magee [16], who introduced the notion of quiescence as a sufficient and necessary condition to ensure state consistency. Although they have proved that quiescence is reachable and sufficient for ensuring state consistency, their approach causes serious disruption in the application that is being updated due to the large number of nodes that need to be passivated.

In this paper, we have overcome this drawback by using the notion of tranquillity. By exploiting the properties of black-box nodes, tranquillity is able to separate subtransactions from their cause. Although not guaranteed to be reachable, tranquillity can be achieved quickly in the majority of cases. We have shown that tranquillity—when reached—is a sufficient condition for ensuring state consistency. In the few cases where tranquillity cannot be reached in bounded time, a fallback mechanism to quiescence can easily be implemented.

The advantages of tranquillity over quiescence are twofold. First, tranquillity has a much smaller disruption than quiescence since a node in a tranquil status does not require all of its adjacent nodes to be in a passive status. Second, tranquillity allows the replacement of nodes at times when it is semantically correct to do so, even when the quiescence condition does not hold.

We have shown that tranquillity can be implemented on top of existing component frameworks. When a fallback to quiescence is required, transparent message tagging can be used in order to avoid breaking the black-box nature of nodes.

## Acknowledgments

Yves Vandewoude and Peter Ebraert are sponsored by a scholarship from the Institute for the Promotion of Innovation through Science and Technology in Flanders.

## References


[1] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin, "Language Support for Connector Abstractions," *Proc. 17th European Conf. Object-Oriented Programming (ECOOP '03)*, pp. 74-102, 2003.

[2] Y. Berbers, P. Rigole, Y. Vandewoude, and S.V. Baelen, "Components and Contracts in Software Development for Embedded Systems," *Proc. First European Conf. Use of Modern Information and Communication Technologies*, pp. 219-226, 2004.

[3] G. Bieber and J. Carpenter, "Introduction to Service-Oriented Programming," www.openwings.org/download/specs/ServiceOrientedIntroduction.pdf, Sept. 2001.
[4] S. Bohner and R. Arnold, *An Introduction to Software Change Impact Analysis,* pp. 1-26. IEEE CS Press, 1996.
[5] A.L. de Moura, C. Ururahy, R. Cerque, and N. Rodriguez, "Dynamic Support for Distributed Auto-Adaptive Applications," *Proc. Second ICDCS Int'l Workshop Aspect Oriented Programming for Distributed Computing Systems,* vol. 2, 2002.
[6] R.S. Fabry, "How to Design a System in which Modules Can Be Changed on the Fly," *Proc. Second Int'l Conf. Software Eng. (ICSE),* pp. 470-476, 1976.
[7] P. Gahide, N. Bouraqadi, and L. Duchien, "Promoting Component Reuse by Integrating Aspects and Contracts in an Architecture Model," *Proc. First AOSD Workshop Aspects, Components, and Patterns for Infrastructure Software,* 2002.
[8] J. Gerlach and S.V. Baelen, "Run-Time Evolution and Dynamic (Re)Configuration of Components: Model, Notation, Process and System Support," technical report, Katholieke Universiteit Leuven, 2003.
[9] H. Goullon, R. Isle, and K.-P. LShr, "Dynamic Restructuring in an Experimental Operating System," *Proc. Third Int'l Conf. Software Eng. (ICSE),* 1978.
[10] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques.* Morgan Kauffman, 1993.
[11] D. Gupta, "On-Line Software Version Change," PhD dissertation, Dept. of Computer Science and Eng., Indian Inst. of Technology, Kanpur, Nov. 1994.
[12] D. Gupta, P. Jalote, and G. Barua, "A Formal Framework for Online Software Version Change," *IEEE Trans. Software Eng.,* vol. 22, no. 2, pp. 120-131, Feb. 1996.
[13] M. Hicks, "Dynamic Software Updating," PhD dissertation, Univ. of Pennsylvania, 2001.
[14] C.R. Hofmeister, "Dynamic Reconfiguration of Distributed Applications," PhD dissertation, Univ. of Maryland, College Park, 1993.
[15] A. Ketfi and N. Belkhatir, "Dynamic Interface Adaptability in Service Oriented Software," *Proc. Eighth Int'l Workshop Component-Oriented Programming,* 2003.
[16] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Trans. Software Eng.,* vol. 16, no. 11, pp. 1293-1306, Nov. 1990.
[17] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 260-267, 1988.
[18] A. Rausch, "Software Evolution in Componentware—A Practical Approach," *Proc. 12th Australian Software Eng. Conf.,* 2000.
[19] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, Jan. 1998.
[20] D. Urting, S.V. Baelen, T. Holvoet, P. Rigole, Y. Vandewoude, and Y. Berbers, "A Tool for Component Based Design of Embedded Software," *Proc. 40th Int'l Conf. Technology of Object-Oriented Languages and Systems (Tools Pacific '02),* Feb. 2002.
[21] Y. Vandewoude, "Dynamically Updating Component-Oriented Applications," PhD dissertation, Katholieke Universiteit Leuven, 2007.
[22] Y. Vandewoude and Y. Berbers, "Semantically Sane Component Preemption," *Proc. ERCIM Workshop Software Evolution,* Apr. 2006.
[23] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "An Alternative to Quiescence: Tranquility," *Proc. 22nd IEEE Int'l Conf. Software Maintenance (ICSM '06),* Oct. 2006.
[24] A. Wils, P. Rigole, Y. Berbers, and K.D. Vlaminck, "Ambient Computing Using Component Resource Contracts," *Proc. IASTED Conf. Advances in Computer Science and Technology,* 2004.

**Yves Vandewoude** received the master's degree in computer engineering, the bachelor's degree in applied economics, and the PhD degree in computer science from the Katholieke Universiteit Leuven (KULeuven) in 2001, 2006, and 2007, respectively. He is a postdoctoral researcher in the Distributed Systems and Computer Networks (DistriNet) Research Group, Department of Computer Science at KULeuven. The focus of his research is in the field of software evolution in general and dynamic software evolution in particular. His other fields of interest include component-based software engineering, middleware, software architecture, and distributed systems. The work described in this paper is part of his PhD thesis and was funded by a doctoral scholarship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen).

**Peter Ebraert** received the licentiate diploma in computer science in 2001, the European master in object, component, and aspect-oriented software engineering technologies, specializing in object-oriented technologies, in 2003, and the diploma in economics from the Vrije Universiteit Brussel, Belgium. He also studied abroad for three separate periods of six months. He is a PhD student and research assistant in the Programming Technology Laboratory at Vrije Universiteit Brussel. His research is in the field of dynamic software evolution and is funded by a doctoral scholarship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT Vlaanderen).

**Yolande Berbers** received the MSc and PhD degrees in computer science, with a dissertation in the area of distributed operating systems, from the Katholieke Universiteit Leuven in 1982 and 1987, respectively. Since 1990, she has been an associate professor in the Department of Computer Science. She was an invited professor at the University of Kinshasa, Zaire, in 1988, at the Franco-Polish School of New Information and Communication Technologies, Poznan, Poland, in 1995 and 1996, and at the Université Pierre en Marie Curie (Paris 6) in 2006. Her research interests include software engineering for embedded software, ubiquitous computing, service architectures, middleware, real-time systems, component-oriented software development, distributed systems, environments for distributed and parallel applications, and mobile agents. She has more than 40 scientific publications in the last five years and has been serving on the program committees of several conferences and workshops. She is a member of the IEEE.

**Theo D'Hondt** received the PhD degree in sciences from the Vrije Universiteit Brussel in 1976, where he is currently a full-time faculty member in the Computer Science Department, Faculty of Sciences. He is responsible for the Programming Technology Laboratory, a software and language engineering research laboratory that was founded approximately 20 years ago. In 1998, he cofounded the European master in object, component, and aspect-oriented software engineering technologies: a joint master of science program between the Vrije Universiteit Brussel, the Ecole des Mines de Nantes, and several other international partners. He was the organizing chair of the 12th European Conference on Object-Oriented Programming (ECOOP '98) and has been a member of the ECOOP program committee since then. From 2004 to 2007, he was the dean of the Faculty of Sciences, Vrije Universiteit Brussel. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.