QoS Assurance for Dynamic Reconfiguration of Component-Based Software Systems

Wei Li

Abstract—A major challenge of dynamic reconfiguration is Quality of Service (QoS) assurance, which is meant to reduce application disruption to the minimum for the system's transformation. However, this problem has not been well studied. This paper investigates the problem for component-based software systems from three points of view. First, the whole spectrum of QoS characteristics is defined. Second, the logical and physical requirements for QoS characteristics are analyzed and solutions to achieve them are proposed. Third, prior work is classified by QoS characteristics and then realized by abstract reconfiguration strategies. On this basis, quantitative evaluation of the QoS assurance abilities of existing work and our own approach is conducted through three steps. First, a proof-of-concept prototype called the reconfigurable component model is implemented to support the representation and testing of the reconfiguration strategies. Second, a reconfiguration benchmark is proposed to expose the whole spectrum of QoS problems. Third, each reconfiguration strategy is tested against the benchmark and the testing results are evaluated. The most important conclusion from our investigation is that the classified QoS characteristics can be fully achieved under some acceptable constraints.

Index Terms—Change management, componentware, dynamic reconfiguration, modeling the QoS assurance process, system evolution.

1 Introduction

Previous authors have highlighted the importance of reconfiguration of software systems to fix bugs, improve performance, and extend functionality [20], [34], [36]. *Static* approaches require shutting down, recompiling, and rebooting the system. Vandewoude et al. [34] have identified three significant disadvantages with static approaches with respect to Quality of Service (QoS): 1) Shutting down could cause the system to lose accumulated state during the ongoing execution of the system; 2) unavailability is unacceptable for long-lived or mission-critical systems; and 3) unavailability promotes poor self-adaptation.

Dynamic approaches are hence proposed as an alternative to static reconfiguration. In this paper, *Dynamic reconfiguration* is a synonym of runtime evolution [10], the ability to change a system's functionality and/or topology while it is running. Dynamic reconfiguration typically involves changes at runtime via addition, deletion, and replacement of components, and/or alteration of the topology of a component system. The benefit is application consistency and service continuity for the system while it is being updated. The motivation for dynamic reconfiguration comes from two aspects: *adaptativeness* and *high availability*, which are both QoS driven. An adaptive system works in unanticipated environments and frequently adapts its behavior in response to a changing environment. A number of typical distributed adaptation scenarios are outlined by Truyen et al. [33], based on a case study of multiple protocol instant messaging systems. In such systems, distributed caching services, message fragmentation services, and reliability requirements are subject to dynamic adaptation. In order to maintain system responsiveness (a QoS requirement), services should only be deployed or removed under certain conditions, and thus dynamic adaptation is needed. In Dowling and Cahill [9], a self-adaptive application (a 3-in-1 phone) is presented. For mobility, the phone can function as a cordless phone within a Bluetooth network, a cellular phone in a cellular network, and a walkie-talkie within range of another Bluetooth phone. In this application, dynamic adaptation is needed to reconfigure its (software) components to function as different phones to deliver appropriate quality of service, depending on the bit rate of the network connection. Mobile and ubiquitous systems also require dynamic adaptation in the form of system updates while active.

The domain of highly available systems is diverse; for example, Warren et al. [36] listed space station oxygen control systems, telecommunication switching systems, and 24/7 e-commerce systems. Significantly, these are all mission critical. Mitchell et al. [24] present a number of mission-critical multimedia applications, including digital TV studios, distributed surgery, and remote surveillance, all of which are noted as being long lived and frequently reconfigured. Construction and evolution of such applications can be effectively addressed by a dynamically evolvable structure. This preserves system usefulness for extended periods without downtime or decline of QoS.

Kramer and Magee's highly influential work in the late 1980s started research in dynamic reconfiguration [20]. The outcome was to enable safely changing the system at runtime by keeping unaffected components operational while the affected part is suspended. A core enabling concept relating to nodes (software components) is that of *quiescence*, where a node is not involved in a transaction

• *The author is with the Center for Intelligent and Networked Systems and the School of Information & Communication Technology, Central Queensland University, Rockhampton, QLD 4702, Australia. E-mail: w.li@cqu.edu.au.*

Manuscript received 30 June 2010; revised 28 Nov. 2010; accepted 28 Feb. 2011; published online 22 Mar. 2011.
Recommended for acceptance by J. Grundy.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-06-0202.
Digital Object Identifier no. 10.1109/TSE.2011.37.

and will neither receive nor initiate any new transactions. The transaction model can be independent (nonnested) or dependent. When supporting dependent transactions, the definition of quiescence and the change rules remain the same as for independent transactions. However, the passivated set must be expanded to account for dependency. As a result, the size of an affected part depends on transaction models: the more interdependent, the more globally quiescent. In 2007, Vandewoude et al. [34] studied quiescence and noted that it could result in significant disruption to the application being updated. They proposed the notion of *tranquillity* to explore the minimal passivated sets of nodes for reduced disruption on ongoing transactions. However, little research can be found in the current literature to systematically address QoS for dynamic reconfiguration.

This paper argues that the true benefit from dynamic reconfiguration is to minimize application disruption. It is the QoS assurance feature of dynamic reconfiguration that essentially differentiates it from any other static techniques. Otherwise, the poor QoS of a dynamically reconfiguring system has no significant advantage over the unavailability of a statically reconfiguring system.

As reviewed in Section 3, dynamic reconfiguration has been studied for consistency, availability, and coexistence, but little studied for continuity or QoS assurance. The gap between the state-of-the-art and QoS assurance is significant and due to the lack of:

1. A refined study and definition of the whole spectrum of QoS characteristics.
2. Exploration of QoS assurance mechanisms other than consistency and availability.
3. Quantitative benchmarking of related work for a clear understanding of a system's QoS in the face of different reconfiguration mechanisms.

This paper aims at bridging the gap by addressing the whole spectrum of QoS assurance problems in terms of:

1. Definition of the whole spectrum of QoS characteristics through detailed study.
2. Analysis of logical and physical requirements and conceptual modeling of each QoS characteristic.
3. Design of application-independent reconfiguration strategies, and realization of related work by the strategies.
4. Design of a generic reconfiguration benchmark to present the complete QoS problems.
5. Benchmarking the reconfiguration strategies for their QoS assurance abilities.
6. Criteria for quantitative evaluation and comparison of testing results.

Our previous research [23] has identified the limitations of existing work in QoS impact analysis and provided a framework to evaluate dynamic approaches for their impacts on the QoS of running software systems. This paper will focus on the theoretical and technical aspects of QoS assurance for dynamic reconfiguration of component-based software systems.

The major contribution of this paper is the identification of the properties of reconfiguration frameworks, namely, dynamic version management (DVM), reconfiguration timing control, stateless equivalence, and controllability of overheads that are potentially useful for maintaining QoS.

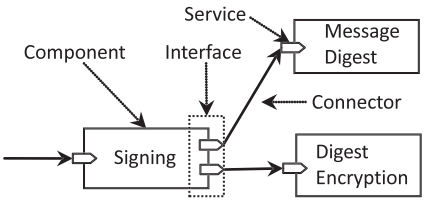

Fig. 1. A component has a provided-interface and a required-interface, indicating services that it offers and needs. A connector links a required-service of a component to a provided-service of another component.

The rest of this paper is structured as follows: As a preamble for the whole paper, an abstract component model is proposed in Section 2 to present components, component systems, and reconfiguration terms. QoS characteristics, key problems, and related work are defined, identified, and reviewed in Section 3. The architectural properties for dynamically reconfigurable frameworks that are treated as potentially useful for maintaining QoS characteristics are proposed and modeled in Section 4. In Section 5, first, reconfiguration strategies are proposed to represent the dynamic reconfiguration approaches (existing or proposed in this paper). Second, a full-featured benchmark to expose the concerned QoS problems is defined. Third, the reconfiguration strategies are tested, and finally, the quantitative comparison of QoS regulation results between our approach and existing approaches are presented. Section 6 concludes by summarizing the achievable QoS characteristics identified.

2 Preamble: An Abstract Component Model and Reconfiguration

To provide a discussion context for dynamic reconfiguration, we first describe the assumptions about components, connectors, and services, and the terms used for reconfiguration. Consequently, we can introduce the dependent concepts in terms of QoS characteristics in later sections.

2.1 Component Systems and Reconfiguration

A *component* is a processing entity that encapsulates a set of functionality and data. Furthermore, a component has a *provided-interface* to specify a set of services that it provides for use by other components. Similarly, a component has a *required-interface* to specify a set of services that it needs. A *service* is a set of functionality, along with the policies that dictate its usage. A *connector* is a directed connection from a *required-service* of a component to a *provided-service* of another component, indicating the client/server relationship between the components. For an individual component, the *client* of its services can be an internal component of the same system or an external software agent, as long as they follow the service specification to make a *request*.

For example, in Fig. 1, the *Signing* component offers a *digital signing* service through its provided-interface and needs two services to function as indicated in its required-interface: *message digesting*, which is here satisfied by

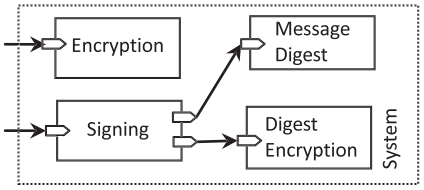

Fig. 2. A component system consists of a set of components linked by a set of connectors. The system as a whole has an external interface, providing services to other component systems or external software agents.

connection to a corresponding service in the provided-interface of the *Message Digest* component, and *digest encrypting*, satisfied by the *Digest Encryption* component.

In providing services to each other, individual components may be assembled into a system, which as a whole may provide services to other component systems or external software agents. A *component system* is assumed to consist of a set of components with directed connections between them by a set of connectors. For example, the components in Fig. 1 can coordinate with an *Encryption* component to form a system that provides an interface with *encrypting* and *digital signing* services as illustrated in Fig. 2.

The proposed component model does not restrict concurrency. For example, in the system of Fig. 2, a client can use two threads, one of which uses the encryption service and the other the signing service for the same block of data. However, if a *compressing* service is needed, the encrypting or signing service should not be required until the data have been compressed. This means that synchronization and concurrency rely on the application rather than the model. Furthermore, the proposed model does not restrict the way to make a request, which could be a request-reply style or a (local or remote) method call.

A *configuration* of a component system is described as the *structural relationship* between components, indicated by the layout of components and connectors. A *reconfiguration* is to modify the structure of a component system in terms of *addition*, *deletion*, and *replacement* of components and/or connectors. For example, to update the example system in Fig. 2 with a newer encryption algorithm, *Encryption* is to be replaced by the new component *Encryption(new)* and the service request is redirected by a newer connector to the new service as illustrated in Fig. 3. Therefore, while the reconfiguration transforms the structural view of a component system, it

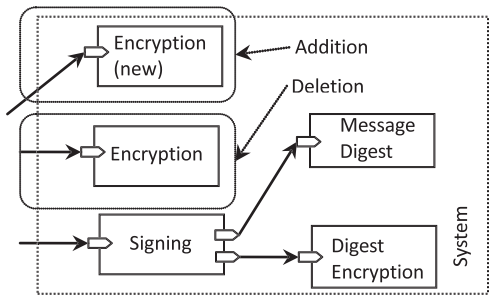

Fig. 3. The functionality of the component system is updated by a reconfiguration via addition and deletion of components and connectors.

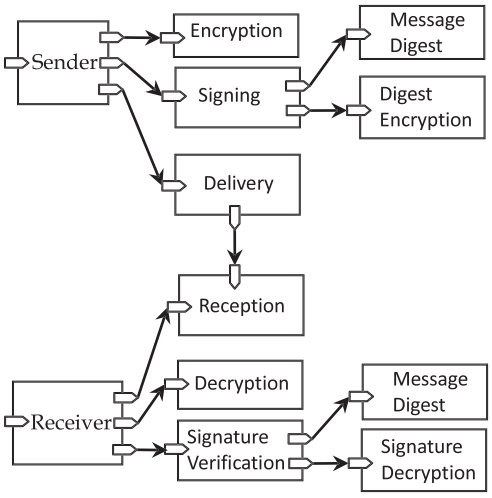

Fig. 4. DEDSS component system uses symmetric and asymmetric encryption and digital signature to ensure the safety of data transmission.

changes the system's functionality and service specification. From another aspect, a reconfiguration may consist of several individual updates or changes to components and/or connectors.

2.2 An Example Component System

To abstract the problems in dynamic reconfiguration and present the corresponding terms, the example in Fig. 2 is extended into a more complex component system as the explanatory context. The system is named *Data Encryption and Digital Signature System (DEDSS)*, which is a secured data transmission system with one *sender* and multiple *receivers*. The security is ensured by *symmetric* and *asymmetric key encryption* and *digital signature*. Fig. 4 illustrates DEDSS with the sender and only one receiver.

The application scenario is: The sender generates a *session key* and uses it to encrypt (*Encryption*) a data-item; the session key is then encrypted by the receiver's *public key*. The same data item is digitally signed (*Signing*): producing (*Message Digest*) the message digest of the data-item and then encrypting (*Digest Encryption*) the message digest with the sender's *private key*. The sender then delivers (*Delivery*) the ciphertext, encrypted session key, and signature in a data-package.

A receiver retrieves (*Reception*) the data-package addressed to it. The receiver decrypts (*Decryption*) the encrypted session key by its own *private key*, and uses the session key to decrypt the ciphertext. To verify (*Signature Verification*) the data-item, the receiver needs to produce (*Message Digest*) a message digest from the decrypted plaintext, and decrypts (*Signature Decryption*) the sender's signature with the sender's *public key*. If the newly created message digest matches the decrypted message digest, the verification is successful, and then the *confidentiality*, *authentication*, and *integrity* of the data item are validated for the transmission.

2.3 Related Concepts

An *application protocol* is a set of behaviors in response to a request. For example, in DEDSS, in response to a request of message transmission, the application protocol performs the encryption and signing before delivery. Similarly, the

application protocol performs the decryption and signature verification after reception.

A component system exhibits more than one protocol in the transformation period from the old into the new subsystem in reconfiguration. A component is *protocol-dependent* if its functional semantics restrict it to work for a single application protocol only. Otherwise, it is *protocol-independent* because it can be shared by the two subsystems and work for any application protocols in the period. In DEDSS, *Delivery* and *Reception* are protocol-independent components, but the rest of the components are protocol-dependent. Furthermore, protocol-dependent components are interdependent because they rely on each other to facilitate the execution of an application protocol. For example, *Encryption* and *Decryption* are dependent components because they must follow an application protocol (a particular encryption/decryption algorithm in this example).

A *protocol transaction* (shortened as *transaction*) is an execution of an application protocol. Furthermore, an application thread is just a runtime representation of a transaction. For example, in DEDSS, the workflow of the execution of the application protocol is as follows, where the arrows, double vertical bars, and square brackets represent sequential, concurrent, or nested requests, respectively:

$$\begin{aligned}\text{Sender} \rightarrow\ & [\text{Encryption} \,|||[\text{Signing} \rightarrow [\text{Message Digest},\\ & \text{Digest Encryption}]],\ [\text{Delivery} \rightarrow \text{Reception}]]\\ \text{Receiver} \rightarrow\ & [\text{Reception},\ \text{Decryption},\ [\text{Signature Verification}\\ & \rightarrow [\text{Message Digest},\ \text{Signature Decryption}]]].\end{aligned}$$

We assume that an application protocol can be implemented by a component system. Therefore, a reconfiguration is a change of the component system from the existing application protocol into another one. *Dynamic reconfiguration* is to accommodate such a reconfiguration without shutting down the operations of the system. For dynamic reconfiguration, the following problems are worth to be investigated.

Suppose that $c_1, c_2, \ldots,$ and c_n are dependent components; $c_1 \rightarrow c_2 \rightarrow \cdots \rightarrow c_n$ is the workflow between them (concurrent and nested requests are omitted for simplicity, which does not affect the following discussion); components $c_1, c_2, \ldots,$ and c_n are to be updated by components $c_1', c_2', \ldots,$ and c_n' by a reconfiguration, and d_i is the last output data item of c_i that is processed for the old protocol, where $i \in \{1, 2, \ldots, n\}$.

The *critical state* of a system is a set of states that are distributed into components $c_1, c_2, \ldots,$ and c_n. In DEDSS, the system's critical state consists of

1. the state of *Encryption*: the receivers' public keys and the session keys already produced;
2. the state of *Decryption*: the receiver's private key;
3. the state of *Digest Encryption*: the sender's private key;
4. the state of *Signature Decryption*: the sender's public key.

A reconfiguration preserves the system's critical state if the state of c_i was transferred into c_i' before the latter is allowed to function, where $i \in \{1, 2, \ldots, n\}$.

A reconfiguration preserves *component dependency* if updating a component c_i, where $i \in \{1, 2, \ldots, n\}$, implies updating all the remaining components c_j, where $j \in \{1, 2, \ldots, n\}$ and $j \neq i$. This update is called a *dependent update*.

A reconfiguration preserves *transaction correctness* if a data item d is processed by either $c_1, c_2, \ldots,$ and c_n, or $c_1', c_2', \ldots,$ and c_n'. A reconfiguration preserves *transaction completion* if it updates c_i after c_i has processed d_{i-1}, where $i \in \{1, 2, 3, \ldots, n\}$, and d_0 is the input into c_1.

An *error status* occurs if:

1. Only some rather than all dependent components are updated. This situation breaks component dependency.

   For example, in DEDSS, *Digest Encryption* and *Signature Decryption* are mutually dependent components. If one is updated, the other must be updated as well.
2. Updating is performed with incorrect timing. This situation breaks either transaction-correctness or transaction-completion.

   For example, consider the scenario in DEDSS, where, while a data-item is signed by the old algorithm via *Signing* but not verified yet, *Signature Decryption* is updated with a newer algorithm, or *Signature Decryption* is simply deleted. Consequently, the data-item is processed by the new algorithm. That leads to an unsuccessful verification. However, the application layer cannot distinguish it from an attacker's interception.
3. The new component functions before the state was transferred in. This situation breaks the system-wide critical state.

   For example, in DEDSS, if the state was not transferred into *Encryption(new)*, it may produce the already-used session keys and therefore reduce the security strength of the system.

A *protocol version* represents a state of a dynamic system in which the workflow of all transactions follows the component-dependency defined by an application protocol. Assume that a system has some version (v_{old}) before reconfiguration. When the new system is introduced, another version is needed (v_{new}). The two different versions can distinguish two workflows, of which one follows the old dependency and the other follows the new dependency. We also suppose that the system is switched into v_{new} when the new system is installed. After installation, any newly created transactions will follow the new protocol, and therefore old transactions can finalize naturally.

When considering concurrency between a reconfiguration and ongoing transactions, the following concepts are necessary for further discussion.

CPUs are *nonsaturated* if their usage is lower than 100 percent on average for servicing application transactions. Otherwise, CPUs are *saturated*.

A threading policy is *preemptive* if a higher priority thread always preempts the execution of a lower priority thread. Under such a policy, if a thread with higher priority has not finished, threads with lower priorities will have to wait. However, threads with the same priority will have the same chance to compete for resources.

A threading policy is *time sliced* if it divides time into schedulable time slices, of which each time slice is further divided into schedulable time slots.

A *reconfiguration planner* parses the configuration specifications of the existing (old) and target (new) system and compares their difference to create an operational reconfiguration plan. A *reconfiguration scheduler* schedules the concurrent execution of a reconfiguration plan with the ongoing transactions. A reconfiguration *strategy* is a combination of a reconfiguration planner and a reconfiguration scheduler. Given a reconfiguration task in terms of the existing and target system's configuration specifications, a *reconfiguration manager* creates and performs a reconfiguration strategy to do the task.

3 QoS Characteristics, Key Problems, and Related Work

Having defined an abstract component model, along with related reconfiguration concepts, we are able to introduce the dependent concepts in terms of QoS characteristics and key problems, with related work reviewed according to each QoS characteristic. The alignment of the QoS characteristics is in line with their strength, which means: 1) A QoS characteristic is stronger than all its previous ones and 2) to support a QoS characteristic, its previous QoS characteristics must be supported in advance. In addition, although related work in the same classified group varies in technique, we assume that the variation is essentially trivial in terms of the defined QoS characteristics.

3.1 Consistency

3.1.1 Definitions

Definition 1. *A reconfiguration is consistent if it preserves:*

1. *the system-wide critical state,*
2. *component-dependency,*
3. *transaction-correctness, and*
4. *transaction-completion.*

In DEDSS, suppose that *Encryption*, *Decryption*, *Digest Encryption*, and *Signature Decryption* are to update. A reconfiguration is consistent if:

1. The states of the components have been transferred into the corresponding new components before their functioning.
2. If *Encryption* is updated, *Decryption* must be updated as well. The same is for *Digest Encryption* and *Signature Decryption*.
3. There is no interference between the old and new workflows during reconfiguration.
4. Any transactions already started by the old protocol must be committed.

3.1.2 Related Work

Kramer and Magee [20] proved that *quiescence* is a sufficient condition to preserve application consistency for updating a running system. Quiescence is a status in which a node (software component) is both passive and has no outstanding transactions. A quiescent node is therefore both consistent and frozen for a safe update. Since this important work, quiescence has become the de facto standard to provide application consistency for dynamic reconfiguration. (There have been many prototype implementations [2], [6], [8], [9], [10], [15], [18], [19], [26], [27], [32], [33], [35], [39] of dynamically reconfigurable systems in which quiescence is the basis of application consistency preservation.) A review of selected work follows.

Ajmani et al. [2] proposed modular upgrades of distributed systems to fulfill the requirements: coexistence of different versions of components and communication using incompatible protocols, legacy behavior retirement, state transfer, automated update deployment, controlled deployment of updates at appropriate times, and limiting of service disruption. To address the issues, an upgrade has six elements:

1. the old class to be replaced,
2. the new class to be introduced,
3. a state transfer function,
4. an upgrade scheduling function,
5. a past component, and
6. a future component to simulate the old class and the new class behaviors, respectively, to address interoperation across versions.

It is evident that the version management is heavy and complicated in terms of redundancy (too many long term coexisting versions: the old, the current, and the future), dynamic call labeling and forwarding, and maintenance of multiversion component library. To preserve application consistency, the approach needs to shut down a node before changing its codes.

In Bidan et al. [6], the preservation of local node consistency is modeled as Remote Procedure Call (RPC) integrity—all RPCs initiated by activities affected by reconfiguration should be completed before the changes. Consistency is assured by applying quiescence to the links and/or components affected by reconfiguration. System-wide application consistency is not considered.

In De Palma et al. [26], local consistency is related to the inner computation of a component, where naming and status of communication must be preserved. Global consistency is concerned with globally distributed transactions, where the completion of a transaction must be ensured. The principal argument is: If a component involves both a reconfiguration transaction and an application transaction, conflict occurs. To resolve the conflict, the application transaction must be rolled back. This isolation algorithm requires that transactions be ordered with time stamps. Such an assumption is questionable due to a lack of global clock in distributed environments.

Rasche and Polze [27] proposed a formal definition of reconfiguration correctness: structural integrity; mutually consistent states of components. They provided an algorithm to ensure reconfiguration correctness at runtime. The strength of the algorithm is to support reentrant method invocations and state transfer. However, reconfiguration concerns are not separated from application logic. To identify ongoing method invocations, a counter must be inserted into the application program. To apply quiescence, the synchronization constructs must also be inserted into the application program.

The main concern of Warren et al. [36] is application consistency, defined as conformance to a set of architectural

constraints. They presented a method for automated verification of whether constraints can hold under reconfiguration. In the article, architectural constraints are expressed as Alloy [16] (an object modeling language based on first-order logic) assertions. A simple case study of a network of routers is given. In the case study, the assertions of *no circle* and *multipath* are discussed. The default mechanism of consistency preservation is quiescence. However, they did comment on the importance of modeling and maintaining performance requirements.

Zhang et al. [39] modeled global consistency as component dependency, and the safety of adaptation was defined as the absence of violation of dependency and interruption on atomic communication (interaction within a component or between components). To demonstrate dependency between components, the encoding and decoding chain was used as a case study. This approach requires that part of the system be *blocked* in order to ensure that the last packet processed by the encoder can be decoded by the corresponding decoder. Then, it is safe to swap the system to use the new encoding and decoding chain.

The most recent articles on dynamic reconfiguration include Guay et al. [11], Leger et al. [21], Surajbali et al. [31], and Wurthinger et al. [37] and [38]. The common feature of these approaches is still to apply quiescence to preserve reconfiguration consistency, while individual approaches use different reconfiguration operations for achieving a quiescent status. For example, Surajbali et al. used *quiescence* operation to put the system into a quiescent status before weaving/binding aspects; Wurthinger et al. used *suspend* and *resume* operations and, in between, the update of Java instances can be done safely.

3.2 Availability

3.2.1 Definitions

Definition 2. *A reconfiguration preserves availability if the system under reconfiguration keeps its servicing online when accommodating the changes.*

The *blocking* operation [20] is the original technique proposed to keep the system online during updating. Blocking maintains a FIFO queue of incoming requests for the affected part of the system. Theoretically, the queue length is infinite so that queued requests are not dropped. Consequently, the system's consistency can be preserved, and at the same time the unaffected part of the system can still function.

In DEDSS, if *Encryption* and *Decryption* are updated, an encryption request can be blocked until the update of *Encryption* is completed. Similarly, a data item that is encrypted by the new algorithm is delayed for decryption before all the data items that are encrypted by the old algorithm are decrypted by the old *Decryption*, and the new *Decryption* can be put in use.

3.2.2 Related Work

The following are typical examples of availability in the literature.

Janssens et al. [18] exploited Aspect-Oriented Programming (AOP) to support a coordination protocol for safe dynamic weaving/binding of distributed aspects. They explained that, in essence, aspects do not fundamentally differ from components, but use a different binding structure. They investigated to what extent NeCoMan's [17] reconfiguration mechanism can be reused for aspect-oriented weaving and unweaving, particularly focusing on the structural integrity, application consistency, and system-wide critical state. However, this work inherited the fundamental property from NeCoMan: applying quiescence to preserve application consistency.

In Kim and Bohner [19], blocking operations are used to drive components to a safe status for reconfiguration. To deal with dependent updates, the model requires that the replacing component must provide the same interfaces as those of the replaced component. The paper models dependent updates, cyclic method calls, and state transfer using an aspect-oriented approach.

The following work went further than just applying quiescence by exploiting a variety of mechanisms to minimize application disruption.

A recent important work to address availability is by Vandewoude at al. [34], in which the concept *tranquillity* is proposed. Tranquillity is an improvement over quiescence because it significantly reduces disruption on the running program. Tranquillity makes use of a minimal passivated set of nodes to minimize the affected area of disruption by quiescence. Tranquillity is still a sufficient condition for application consistency. Tranquillity can occur naturally, but once it holds for a node, all interactions between the node and its environment must be *blocked* in order to keep the node in a tranquil status for the duration of update. Tranquillity is not proven to be reachable in bounded time. As a result, a dynamic reconfiguration system based on tranquillity must also use quiescence as a backup.

The dynamic reconfiguration service of Bidan et al. [6] is built on the semantics of the LifeCycle CORBA service. The algorithm introduces minimal disruption to system's execution. The idea of minimization of disruption is similar to tranquillity [34] because disruption is restricted to only the part of system affected by reconfiguration.

In designing reconfigurable architecture, Dowling and Cahill [8], [9] separated architectural concerns from functional concerns and provided an architectural metamodel to build dynamic software systems via architectural reflection. They presented a typical reconfiguration protocol for maintaining availability [8]. Stateful components and state transfer are supported by their model, but the protocol must *freeze* computation in components involved in reconfiguration to manage system integrity and consistency. The algorithm tries to reduce the length of reconfiguration phase and allows concurrent client invocations on components that are not frozen.

The language-based framework focuses on evolving the contents of the system in a single address space, whereas the architectural approach focuses more on evolving the system at a higher architectural level through replacing individual components and altering topology. Evans [10] addressed reconfiguration to bridge the gap between the two approaches by exploiting the advantages of both. In the proof-of-concept implementation DRASTIC, a distributed

application is divided into zones. A zone is the unit to evolve, and the evolution of a single zone involves temporarily *suspending* the activities between the zone and other contracted zones. Application consistency is preserved for terminated zones, and other zones can still be active for service availability.

For maintaining service availability, the overall objective of De Palma et al. [26] is to block only the part of the application affected by the modification. Their goal is to minimize the overall penalty on the running applications. However, the potential rollback of application transactions in their algorithms reduces system performance.

Zhang et al. [39] concentrated on availability in the face of reconfiguration. The approach blocks the initialization of newer transmissions, but uses a safe adaptation graph to find the minimum safe adaptation path (MAP). Given the cost (packet delay) of each adaptation operation, the MAP is the sequence of operations to change the system from source configuration to the target configuration with minimum blocking time. The drawback of the approach is that determining costs of adaptation operations is application dependent and sometimes there might not be alternatives to find a MAP.

3.3 Coexistence and Continuity

3.3.1 Definitions and Key Problems

Definition 3. *A reconfiguration possesses coexistence characteristic if the new subsystem is brought into effect fully by the reconfiguration before shutting down and then removing the old subsystem.*

The coexistence characteristic provides reconfiguration with the ability to avoid the use of blocking-like operations. This ability supports a further QoS characteristic defined as follows:

Definition 4. *A reconfiguration possesses continuity if client requests submitted to the running system under reconfiguration are never logically blocked by the reconfiguration.*

While the coexistence characteristic makes full use of concurrency to totally remove the blocking operations, it raises a new problem: *partially shared structure*. During the coexistence period, the old and new subsystems are not structurally independent. They temporarily share some components and the connectors between them. For example, in DEDSS (Fig. 4), to update *Encryption* and *Decryption*, the old application protocol will share *Delivery* and *Reception* with the new application protocol in the coexistence period. In workflow, however, the two subsystems must be independent and must not interfere with each other because they work for different application protocols. To provide continuity, the challenge is: *ensure independent workflow in partially shared structure*. Our solution to this problem is *dynamic version management*, which is presented in Section 4.1.1.

3.3.2 Related Work

Some related work aims to provide for the coexistence of new and old application protocols [7], [10], [13], [28], [32] and therefore enables version compatibility during reconfiguration. Other work [18], [24], [33] goes further to exploit coexistence characteristics to avoid blocking operations and therefore logically maintain service continuity.

Cook and Dage [7] explored coexistence of multiversion components for enhancing system-level reliability. Instead of removing old components immediately, multiple versions of components may remain running temporarily. All versions of the components process the same requests and a voting scheme decides which versions are correct. The new version is kept if it produces correct results. Otherwise, upgrades are rolled back. The mechanism uses *arbiter* components to provide indirection—a single image of component versions. However, they did not further explore coexistence for service continuity in the face of reconfiguration.

Senivongse [28] supported dependent upgrades but permitted unsynchronized evolution of dependent components. Evolution of a new client can be delayed, and the old version client and the new version service may coexist. The *mediator* component provides cross-version interoperability. This work aims at providing evolution transparency by exploiting coexistence. Senivongse did not explore use of coexistence for supporting the continuity of services.

In Evans [10], Hicks et al. [13], and Tewksbury et al. [32], coexistence is exploited for providing compatibility between the old client and the new service. Dependent upgrades are supported but the client upgrade can be delayed. Coexistence is not further applied to support service continuity.

Janssens et al. [18] noted that coexistence enables logical removal of blocking operations and still ensures application consistency. Activating the new interceptors before driving the old ones to a safe status reduces the service disruption caused by quiescence, and thereby preserves application continuity for the system during reconfiguration. The rationale is that the latter will occur while the new components are already brought into effect. However, their dynamic management of the coexisting versions requires that a component has no external state dependencies on the invoking clients.

Mitchell et al. [24] explored QoS and integrity assurance for the reconfiguration of complex multimedia applications. The structural integrity requirement is that the media formats handled by interconnected components should be compatible with one another. QoS constraints are modeled as application-specific requirements such as frame display rates. The model permits the new components to be started before removing the old components in order to fulfill the QoS requirements. However, because there is no dynamic version management for coexistence, it relies on precise timing to ensure when to stop old components and when to start new components. The scheduling relies on predetermination of the processing latency of each component and the startup time of each new component. To some extent, their work exploits coexistence for QoS assurance. However, they identified the following limitations:

1. It is questionable if resource requirements can always be modeled accurately as piecewise linear or quadratic functions.
2. It is not clear if the model is generic enough to be reused to other applications in different domains.

3. The timing-based scheduling algorithm is highly reliant on the predetermined temporal constraints of components and is fully developed for the single path reconfiguration only.

DyReS [33] is a typical algorithm to maintain availability because it applies quiescence to preserve system consistency. The idea of maintaining the performance of a running system under reconfiguration is achieved through the customization of the generic reconfiguration algorithm. To customize the algorithm, the framework requires some application-specific characteristics. For example, switching the order between finishing the old and activating the new service is only possible when the service has no external state dependencies on invoking components. To ensure a noninterleaving coexistence of two versions, the algorithm relies on the *marker* and *dispatcher* components. The marker is responsible for attaching versions to messages and the dispatcher forwards messages to the correct services. When the introduction of coexistence reduces application disruption, however, dynamic installation of marker or dispatcher still relies on *interrupt* operations, which degrades QoS directly. While coexistence is identified as important for QoS, the contribution to the continuity characteristic is application dependent and therefore limited.

Redundancy is a typical approach to providing continuity by hot standbys. However, redundancy is expensive in terms of additional hardware and software that incur additional costs. In addition, redundancy adds extra complexity to maintain the consistency between the active primary replicas and the upgraded secondary replicas [13].

3.4 QoS Assurance

3.4.1 Definitions and Key Problems

While logical continuity is necessary to avoid disturbance to ongoing transactions, it is not sufficient to assure QoS physically. To explain this, we describe the problem of unintentional blocking.

Definition 5. *A reconfiguration causes unintentional blocking to ongoing transactions if it makes resources needed by ongoing transactions unavailable.*

One reason for unintentional blocking is from the preservation of the system's critical state, i.e., transfer of states between the old components and their replacements. There are two factors: time to transfer state and state size.

Time to transfer state. Suppose that at time t_s, the system is switched from version v_{old} to version v_{new}, and at time t_c, all transactions of version v_{old} are committed, where $t_s \leq t_c$. Existing stateful systems [2], [8], [13] achieve state transfer at t_c. The rationale is that when components are no longer used by any transactions of version v_{old}, state transfer is safe in terms of application consistency. Otherwise, early transfer of state may disable some components and result in transaction failure. However, a particular component's state could be available for transfer at any time t_a: $t_s \leq t_a \leq t_c$. The reason is that depending on the data being processed, the component is probably no longer invoked in the whole period from t_s to t_c. On the other hand, in a multithreading environment, the progress of a transaction of version v_{new} could go ahead of a transaction of version v_{old}. Consequently, the replacing component is likely already waiting for the state between t_s and t_c, However, the state is delayed for transfer at t_c due to the requirement of consistency preservation. State transfer at t_c is safe but too conservative and could cause unintentional blocking. In conclusion, we cannot predict when to serialize a stateful component (and deserialize the state into a new version) such that unintentional blocking does not occur. This is because architectural concerns and functional concerns should be separated, and therefore the inner functionality of a component and data being processed should not be inspected by the reconfiguration framework. Our solution to this problem is reconfiguration *timing control*, which is presented in Section 4.1.2.

State size. Because the size of state varies from component to component, state transfer by a deep copy of state variables sometimes needs considerable CPU time and causes threads to wait, i.e., an unintentional blocking. Our solution to that problem is *state sharing*, which is presented in Section 4.2.1.

Furthermore, the combination of state sharing and timing control enables a reconfiguration to have the stateless equivalent property as defined below.

Definition 6. *A stateful system can be regarded as stateless equivalent for reconfiguration if state transfer does not incur unintentional blocking of workflow.*

Another cause of unintentional blocking is reconfiguration overhead, which could physically degrade QoS considerably. When reconfiguration executes concurrently with ongoing transactions on the same physical machine, it can suspend the ongoing transactions due to temporary occupation of resources (CPUs, memory, or network bandwidth). The runtime overheads of reconfiguration have been stated in the current literature [2], [10], [12], [13], [24], [27], [33], but little solution has been suggested from them. Our solution to this problem is *preemptive* scheduling and *time-slice* scheduling, which are presented in Section 4.2.2.

Definition 7. *A reconfiguration provides QoS assurance if the running system's QoS is physically maintained by the reconfiguration not lower than a predetermined baseline in the duration of reconfiguration.*

To provide QoS assurance, a reconfiguration must have full control over unintentional blocking and reconfiguration overheads. Therefore, even if it is impossible to always maintain resource availability for ongoing transactions, it can ensure resource availability with certainty. In conclusion, QoS assurance is a comprehensive utilization of the proposed solutions so far, which will be detailed in Section 4.

3.4.2 Related Work

Some existing work shows awareness of the importance of QoS assurance via their article titles but their investigation and outcomes are very limited.

In Almeida et al. [3], general issues about system QoS during dynamic reconfiguration are discussed. The paper simply presents availability as an important QoS characteristic. Noteworthy properties include: the percentage of time that the system is functioning without disruption from

reconfiguration, the mean time between disruptions, and the mean time to repair. They argued that it is important for dynamic reconfiguration to fulfill availability constraints. They discussed the importance of availability characteristic and how to avoid downtime. However, coping with the dynamic QoS metrics, like throughput and response time, is not discussed.

Gorinsek et al. [12] stressed the importance of maintaining QoS when reconfiguring systems. They indicated that reconfiguration itself requires a significant amount of resources and easily causes the running system to violate its timing or memory constraints. Therefore, they modeled the management of QoS by predetermining the resource consumption of the new components or reconfiguration itself. In their work, the impact of the given reconfiguration is evaluated before actions. Among others, timing constraints and resource consumption are modeled by component contracts, intercomponent contracts, and update contracts. The system is queried about available resources. When resource requirements do not exceed available resources, the update is queued for execution. The real consumption of resources is monitored during the update. The update may be aborted if violation of the constraints is identified. Unless a real resource enforcement mechanism is implemented to control resource consumption required by reconfiguration, the QoS management of Gorinsek et al. is static because it simply rejects or schedules an update to minimize potential disruption on the running system based on resource contracts and predetermined resource consumption.

4 Architectural Properties for Dynamically Reconfigurable Systems

In this section, we propose and model the architectural properties for reconfiguration frameworks that are potentially useful for maintaining QoS characteristics that are defined in Section 3.

4.1 Modeling Service Continuity

4.1.1 Dynamic Version Management

To support all QoS characteristics up to continuity, we propose *dynamic version management* in Section 3.3.1, which will be discussed in detail in this section. As discussed in Section 2, two versions (v_{old} and v_{new}) are sufficient for version management of protocol-dependent transactions. DVM uses the following elements as the version carriers. The *system* as a whole is versioned; both *connectors* and *application threads* (representing transactions) are versioned. On this basis, DVM is designed to apply versioning to three aspects: *shared structure*, *independent workflow*, and *timing*.

The feature of *independent workflow on partially shared structure* is fundamental to DVM. Structurally, in Fig. 5, versioned connectors permit that a required-service can be connected to two provided-services (called a *bi-link*), of which one is the old service to be removed and the other is the new service just added. Consequently, the versioned connectors are to model a partially shared structure between the old and the new subsystem. While a component can be shared by the two subsystems, the versioned connectors are able to distinguish the two networks of components during coexistence. In workflow,

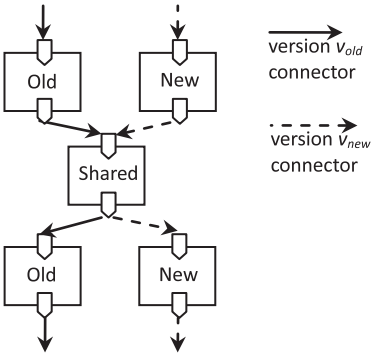

Fig. 5. The new and old application protocols are distinguished by two invocation chains. At the shared component, an application thread makes dynamic selection of an invocation chain by matching its own version with a connector's version. The system determines timing (when starting the new version or stopping the old version).

a versioned *application thread* just works for a single protocol: either v_{old} or v_{new}. If a required service is linked to a unique provided service by a single connector, the version of the connector is ignored by the thread. In this case, the application thread just follows the connector to invoke the service. However, if there are bi-links, the thread makes the decision, *choosing the connector whose version matches its own version*. Under such versioning, the partially shared structure supports two independent invocation chains, of which one works for the old protocol and the other works for the new protocol. These two invocation chains do not interfere with each other. Furthermore, because a new protocol can accommodate any number of protocol-dependent components, the feature supports dependent upgrades naturally.

A further explanation of the versioned structure explores the existence of bi-links. If bi-links exist in the structure, the system is in the coexistence period and versioning is important to ensure independent workflow. Otherwise, if there is no bi-link, the system is in normal execution, versioning is not important, and the structure works for a single protocol. For example, in the absence of bi-links, the application threads or connectors can have either v_{old} or v_{new}, but the system is regarded as working for a single protocol. In that situation, the connectors' versions are ignored by the application threads as there exists only a one-to-one relationship from required services to provided services. This explanation is helpful for an understanding of the timing control of DVM detailed below.

4.1.2 Timing Control

As discussed in Section 3.4.1, while the independent workflow is able to ensure that a data item is being processed by only one protocol, timing is important to ensure that the workflow is not physically blocked. We discuss the details of reconfiguration timing control in this section. Referring to the dependent update example of DEDSS in Section 2.3, the timing must guarantee that when the ciphertext encoded by the new algorithm arrives, the new decoder is ready for decoding, that is, there must be a nonblocking workflow in the chain from the encoder to the decoder.

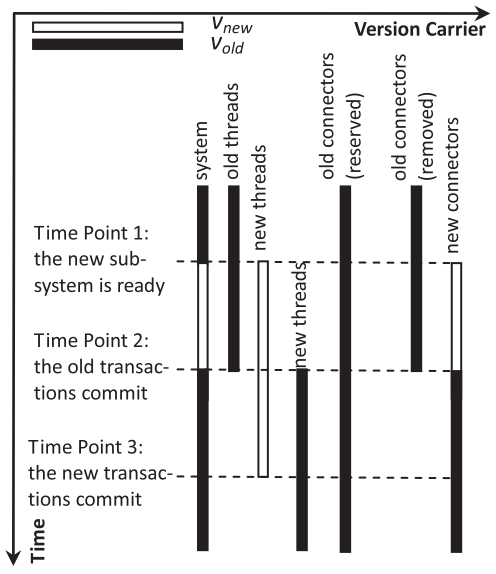

Fig. 6. The system version is the global clock for timing control of DVM. The first two time points mark the start and finish of the coexistence period, respectively. The third time point marks the end of reconfiguration.

The system's version is used as the global clock for timing control. First of all, suppose that the system keeps running in version v_{old}. In this situation, all the connectors have version v_{old} and all newly created application threads have v_{old}. (A thread's version remains unchanged throughout its life cycle.) For a system running in this situation, reconfiguration can start at any time. The first task of reconfiguration is to make (i.e., to load and initialize the new components, add the new connectors, and set the version of the newly added connectors to v_{new}) the new subsystem standby.

For a reconfiguration that supports the coexistence property, there are three critical *time points* for applying timing control, as illustrated in Fig. 6.

The first time point is when the new subsystem is fully installed and ready for execution. At this point, the system switches from v_{old} into v_{new}. If an application thread is created after this time point, it is marked v_{new} because a newly created thread is always marked with the current system's version.

At the second time point, all transactions marked v_{old} are completed, and there are only transactions marked v_{new} remaining in the system. At this time point, all the connectors marked v_{old} must be removed if they are not shared by the new subsystem. After this removal of bi-links, a single required service can only be linked to a single provided service, and the whole system is considered working for the new protocol only. To return the system to normal execution and be ready for future reconfiguration, all connectors marked v_{new} are then marked v_{old}. After that, the system's version returns to version v_{old}. As a result, any newly created application threads after this time point will be marked v_{old} again. The already initiated v_{new} transactions keep going correctly (because there is no bi-link) till all of them commit by the third time point.

At the third time point, there are only transactions marked v_{old} and connectors marked v_{old}. This point marks the end of reconfiguration, and the system is back to normal execution and ready for future reconfiguration. Checking the system, connectors, and threads in Fig. 6, they are marked v_{old} before *Time Point 1* and back to v_{old} after *Time Point 3*. DVM ensures independent workflow between *Time Point 1* and *Time Point 3* when the old and new subsystems coexist.

The timings for the system, thread, and connector versioning are coordinated to permit reconfiguration to occur at well understood time points. This feature ensures: 1) full effectiveness of the new subsystem before safe removal of the old subsystem, and 2) the safe coexistence of old and new subsystems during reconfiguration. Therefore, continuity is achieved by DVM through timing-controlled independent workflow in a shared structure.

4.2 Modeling QoS Assurance

As discussed in Section 3.4, QoS assurance requires reconfiguration to further possess both stateless equivalence and overhead controllability.

4.2.1 Stateless Equivalence and Constraints

Modeling stateless equivalence has two aspects. First, state should be transferred by redirecting references to state variables from the replaced component into the replacing component. If many objects make up the component's state, these objects can be encapsulated into a single wrapper object. Consequently, state can be transferred by redirecting only a single reference, i.e., the reference to the wrapper object. By such encapsulation and redirection, state size is independent of application and a state transfer can be treated as *virtually instantaneous*.

Second, state sharing between the replaced component and the replacing component is also applicable to the problem of unintentional blocking. State sharing requires the following constraints:

1. The replacing component should provide backward compatibility of the state format for the replaced component.
2. Access to the shared state must be mutually exclusive.
3. The application logic permits state sharing.
4. The replacing and replaced components reside on the same physical machine.

Mutual exclusion is needed for access to the shared state and ensures atomicity of the state update and noncorruption of state. However, whether the state is being used by the old or the new component, processing is contributing to the overall QoS of the system, i.e., maintaining QoS during the reconfiguration. The third constraint implies that state sharing is application dependent. State sharing is not applicable to every application, but we assume it can be used for a variety of applications. Identifier generators or number counters are simple examples of components where application logic permits such state sharing. Although component state migration is an open issue (Section 6), the proposed state sharing is applicable to distributed environments, where components can be distributed as long as the replacing and replaced components reside on the same physical machine. The above constraints can be

TABLE 1
Thread Priorities and Scheduling Policies

Reconfiguration	CPU Saturation		CPU non-Saturation	
Phase	Scheduling Strategy	Prioritized Thread	Scheduling Strategy	Prioritized Thread
installation	time-slice	same	pre-emptive	ongoing transactions
transformation	pre-emptive	reconfiguration	pre-emptive	reconfiguration
removal	time-slice	same	pre-emptive	ongoing transactions

easily fulfilled by many domain applications and are more practical to apply. If state sharing is possible, it can be achieved at the second time point mentioned in Section 4.1.2 for all stateful components. Because state sharing can be achieved instantaneously by redirecting a single reference, state sharing enables all version v_{new} components to standby by the second time point and is able to avoid the unintentional blocking problem discussed in Section 3.4.1. In some circumstances, state sharing may be impossible (if the application does not fulfill Constraint 3) and unintentional blocking could occur. However, if possible, state sharing should be applied, and improved QoS performance can thus be achieved for reconfiguration. We assume that state sharing is possible in most cases.

4.2.2 Controllability of Overheads

To address the controllability of reconfiguration overheads, a reconfiguration should be divided into three sequential phases which group different reconfiguration operations by CPU intensiveness.

1. The *installation* phase performs operations to load and initialize new components (including state sharing) and sets up new connectors.
2. The *transformation* phase switches workflow from version v_{old} to version v_{new}, and traces ongoing transactions of version v_{old} to completion.
3. The *removal* phase finalizes and garbage collects old components and deletes old connectors.

Considering the operations in *transformation* phase, switching workflow versions amounts to resetting some flags in RAM and therefore can be treated as *virtually instantaneous*. Tracing the ongoing transactions to completion involves a *wait* operation. However, using thread synchronization, the reconfiguration thread yields its CPU via the *wait* operation and is awakened when a predefined condition, such as completion of a subtransaction at a component, is fulfilled. The *wait* operation is asynchronous and consumes little CPU time. Under such a classification, overheads of this phase can be ignored and it is safe to regard this phase as virtually instantaneous.

However, both the installation phase and the removal phase involve CPU-intensive operations.

On the basis of such a grouping, the controllability of overheads is achieved through thread prioritization and scheduling in consideration of current CPU usage.

When the CPU is nonsaturated, the best solution to controllability is just to make use of free CPU time to execute CPU-intensive operations. Thus, preemptive scheduling should be applied to both installation and removal phases to ensure that the intensive operations can only use free CPU time for execution and must yield the CPU once an application transaction is ready to execute.

When the CPU is saturated, the reconfiguration thread should be assigned the same priority as for the ongoing transactions in both installation and removal phases so that the intensive operations have a chance to execute. To restrict resource competition from CPU-intensive operations, the reconfiguration thread's CPU usage should be restricted in these phases via time-slice scheduling.

Periodically, a reconfiguration thread is scheduled for execution in the *runnable* time slot and suspended in the *sleeping* time slot. By adjusting the weighting of the runnable time slot, the reconfiguration thread CPU usage is restricted. For instance, if 300 ms are assigned to the runnable time slot, the reconfiguration thread will obtain a 30 percent chance to compete for CPU usage in a 1 s time slice. Under such controlled competition, CPU usage is guaranteed at a certain level for ongoing transactions and thus likewise QoS can be maintained at or above a corresponding baseline.

The operations in *transformation* phase can be made logically instantaneous, as discussed previously in this section. Once the reconfiguration thread enters the transformation phase, it cannot be interrupted. This is because operations in this phase must be completed in one run. Otherwise, the feature of *logical instantaneousness* would be broken and unintentional blocking could occur. To preserve this feature, the reconfiguration thread priority should be raised for the transformation phase. The prioritizing and scheduling policies, in relation to CPU usage, are summarized in Table 1 for the operations in the three phases.

In conclusion, a better solution to QoS assurance, taking into consideration key problems and a review of the state of the art, requires comprehensive application of the whole spectrum of QoS characteristics proposed so far in this paper.

5 Quantitative Analysis of Reconfiguration Strategies

In this section, we first present the implementation of the abstract component model and the reconfiguration strategies by which the related works are realized. Then, we propose benchmarking criteria and a benchmark design of DEDSS, along with scenario settings such that DEDSS can expose a rich set of QoS problems for comprehensive evaluation. We finally present the QoS testing procedure and evaluate testing results.

5.1 Implementing Reconfiguration Strategies in Software for QoS Testing

To make available an evolvable architecture and test QoS characteristics defined in Section 3, we have implemented a proof-of-concept prototype *Reconfigurable Component Model* (*RCM*) in Java. RCM is a full implementation of the abstract component model defined in Section 2. In RCM, components and connectors are implemented as Java classes, and used to assemble a component system. RCM's implementation satisfies three basic requirements. First, the whole spectrum of QoS characteristics has been fully implemented through RCM planners and schedulers. Second, a QoS characteristic or a group of characteristics can be integrated into a reconfiguration via RCM Application Programming Interfaces (APIs). This feature allows reconfiguration strategies representative of those proposed by previous authors onto RCM and enables sound comparison of them in the same testing context. Third, the test environment is parameterized by the required workload for evaluation, and the QoS performance of a running system under reconfiguration can be monitored (via plug-in points) and recorded for any predetermined time period. In addition, RCM incorporates a reconfiguration manager, automating the reconfiguration procedure: inputting configuration specifications, choosing planners and schedulers, setting problem scale and workload, and performing and monitoring reconfigurations.

The limitation of RCM is that it does not support component migration in distributed environments. This is due to the need to apply stateless equivalence, which requires the replacing and replaced components to reside on the same physical machine. We suppose to relax this constraint when stateless equivalence is studied for state migration.

5.1.1 Reconfiguration Planner

In RCM, given the declarative description (in XML schema [22]) of the original and target structures of a system, a reconfiguration planner parses the declaration to create inner configuration specifications (in Java classes) of the system's structures, analyze their differences to create an inner reconfiguration specification (in Java classes), and finally create an operational reconfiguration plan (in Java classes). The reconfiguration operations are so organized that the plan possesses predetermined QoS characteristics. In RCM, there are three built-in planners:

The *availability* planner (`rcm.util.RCMAvailabilityPlanner`) applies quiescence to preserve application consistency. The blocking operation is applied and component state is transferred by a deep copy of state variables.

The *continuity* planner (`rcm.util.RCMContinuityPlanner`) applies DVM to avoid quiescence. (Blocking operations are unnecessary and never used by this planner.) DVM ensures safe coexistence of the old and new subsystems and logical continuity of the system service. A component's state is still transferred by a deep copy of state variables.

The *stateless-equivalence* planner (`rcm.util.RCMStatelessEquivalencePlanner`) still applies DVM. However, component state's variables are encapsulated in a wrapper object, and state-sharing is applied by redirecting the state's reference when doing state transfer.

5.1.2 Reconfiguration Scheduler

In RCM, a scheduler schedules a concurrent execution of a reconfiguration plan with ongoing transactions. A scheduler has a built-in scheduling policy, which is parameterized as necessary. For example, in time-slice scheduling, lengths of time slice and time slot are user parameters. A scheduler is able to start a reconfiguration plan at a specified time. In RCM, there are three built-in schedulers:

The *competition* scheduler (`rcm.util.RCMCompetitionScheduler`) assigns the reconfiguration thread the same execution priority as that of ongoing transactions.

The *preemptive* scheduler (`rcm.util.RCMPreemptiveScheduler`) enables the ongoing transactions to preempt the reconfiguration thread in the installation and removal phases, but reverses their priorities in the transformation phase.

The *time-slice* scheduler (`rcm.util.RCMTimeSliceScheduler`) assigns the reconfiguration thread and the ongoing transactions the same execution priority for the installation and removal phases. The scheduler applies time-slice scheduling to the two phases. In each time slice, the ongoing transactions are always runnable, but the reconfiguration thread is made runnable only for a predetermined time slot.

The design of the last two schedulers corresponds to the thread priorities and scheduling policies in Table 1 in Section 4.2.2.

5.1.3 Reconfiguration Strategies

A reconfiguration *strategy* is a combination of a reconfiguration planner and a reconfiguration scheduler. When a strategy is applied, an operational reconfiguration plan is first created by the planner by taking as inputs the declarative description (in XML schema [22]) of the original structure and the target structure of the system. The scheduler then schedules the concurrent execution of the reconfiguration plan with the ongoing transactions in the specified parameters and at the specified time.

To test the QoS abilities of different reconfiguration mechanisms, a number of reconfiguration strategies have been designed to achieve the classified QoS characteristics. As the strategies are application independent, if an existing mechanism achieves a set of QoS characteristics, it can be realized by a reconfiguration strategy. From another point of view, applying a strategy to a reconfiguration demonstrates the system's performance under the reconfiguration for the given strategy. Therefore, the design of reconfiguration strategies facilitates sound comparison of different mechanisms for QoS assurance in the same scenario.

In RCM, the effective combinations of planners and schedulers are outlined in Table 2. A strategy in the table possesses a certain number of QoS characteristics and therefore realizes a number of existing mechanisms, or is unique to RCM. Of these, *sel-cpt*, *qos-pre*, and *qos-ts* are proposed by this paper, while the last two are our final goals for QoS assurance.

RCM is a flexible and open framework in that it separates reconfiguration algorithms (termed strategies in RCM) out from other aspects and supports the *integration* of different reconfiguration algorithms onto the framework

TABLE 2
Reconfiguration Strategies with Built-In QoS Characteristics

Strategy	Planner and Scheduler	What to Simulate	Realization of Model
avl-cpt	rcm.util.RCMAvailabilityPlanner rcm.util.RCMCompetitionScheduler	availability (quiescence or tranquillity)	[2], [4], [5], [6], [8], [9], [11], [15], [19], [20], [21], [25], [26], [27], [31], [34], [35], [37], [38], [39]
con-cpt	rcm.util.RCMContinuityPlanner rcm.util.RCMCompetitionScheduler	continuity (safe coexistence)	[7], [10], [13], [14], [18], [24], [28], [29], [32], [33]
sle-cpt	rcm.util.RCMStatelessEquivalencePlanner rcm.util.RCMCompetitionScheduler	continuity plus stateless equivalence	RCM
qos-pre	rcm.util.RCMStatelessEquivalencePlanner rcm.util.RCMPreemptiveScheduler	QoS-assurance (applicable to CPU non-saturation)	RCM
qos-ts	rcm.util.RCMStatelessEquivalencePlanner rcm.util.RCMTimeSliceScheduler	QoS-assurance (applicable to CPU saturation)	RCM

through implementing reconfiguration *planners* and *schedulers* as plug-ins. If one wants to exercise a newer reconfiguration algorithm, one implements a newer reconfiguration planner (to control the actual use of reconfiguration operations) and scheduler (to control their execution procedure). This enables introduction of newer algorithms when these emerge with properties not readily suited to the existing algorithms. RCM supports this sort of flexibility by its reflection ability. Once RCM obtains a system's specification, it can reflect the system's structure at runtime to perform a reconfiguration strategy on it. This flexibility, though not detailed in this paper, is discussed in a previous publication [22]. Another benefit of RCM is scalability, because RCM restricts DVM to the necessarily affected parts only. Once the reconfiguration specification is obtained, RCM will identify the necessary parts (termed *restrict segments* in RCM) and perform reconfiguration operations on these parts only. This feature restricts the spreading of reconfiguration to the unnecessary parts. The details of restricting DVM to the restrict segments were presented previously [22]. These efforts aim for an application-independent, reconfigurable, and open framework which supports flexibility and scalability naturally.

5.2 Reconfiguration Benchmark and Scenario Setting

Much of the current literature presents case studies to demonstrate the effectiveness and efficiency of research. Case studies cover a variety of applications, such as web applications [13], [15], [34], multimedia applications [24], [39], distributed adaptation [33], [39], network routing [14], [35], and mobile applications [9], [24]. Among these case studies, there is no a single case which covers the whole spectrum of QoS problems studied in this paper. For example, some case studies are restricted to stateless systems [14], [24], [39], and others do not support dependent updates [9], [13], [15], [34], [35]. Some, such as [13], [15], [18], only use micro cases of simple client/server applications.

The motivation to design a benchmark and to specify scenarios is to comprehensively evaluate research in dynamic reconfiguration, especially with regard to QoS assurance abilities. Consequently, the following design criteria are applied: First, the benchmark should be a stateful application and accommodate dependent updates. Second, the correctness of reconfiguration should be verified by simple criteria with respect to the benchmark. Third, the benchmark should be universal enough to be able to evaluate different reconfiguration mechanisms. Fourth, important features of reconfiguration, for instance, the size of the component's state, CPU load, or communication load, must be parameterized to reflect varying problem size or workload. Then, the impacts of problem scale and workload on QoS or the scalability of the framework can be quantitatively evaluated. Finally, the benchmark should be applicable for monitoring and recording QoS at runtime.

In RCM, the benchmark DEDSS is designed to fulfill the above criteria. The application scenario of DEDSS is presented in Section 2. To benchmark reconfiguration strategies, the following reconfiguration scenario (illustrated in Fig. 7) is defined to cover component *insertion*,

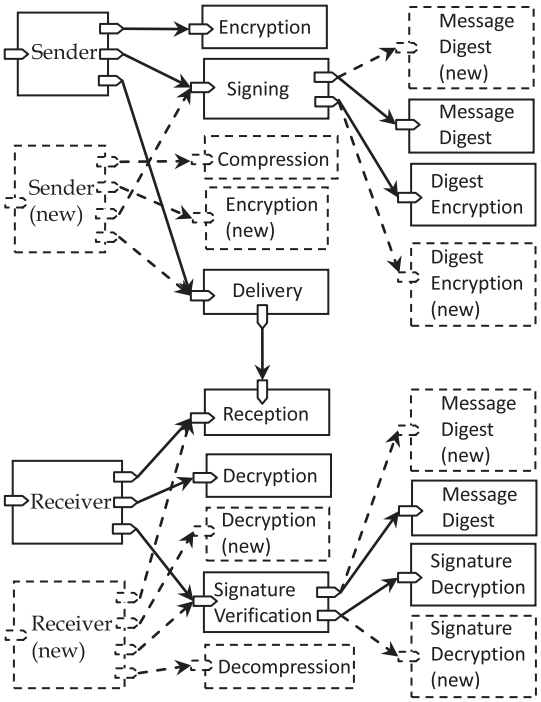

Fig. 7. The original DEDSS is drawn by solid lines. The reconfiguration inserts *Compression* and *Decompression* and upgrades *Encryption*, *Decryption*, *Message Digest*, *Digest Encryption*, *Signature Decryption*, *Sender* and *Receiver* into *Encryption(new)*, *Decryption(new)*, *Message Digest(new)*, *Digest Encryption(new)*, *Signature Decryption(new)*, *Sender(new)*, and *Receiver(new)*, respectively. The upgraded part is drawn by dotted lines.

replacement, and *state transfer*. In particular, this scenario is designed to cover *dependent updates*.

Data compression is a newly required function for the system. Therefore, component *Compression* must be inserted into the sender's subsystem and component *Decompression* must be inserted into every receiver's subsystem. The insertion of *Compression* and *Decompression* is a dependent update.

The algorithm for data encryption/decryption is to be upgraded. Therefore, *Encryption* must be replaced by *Encryption(new)* for the sender's subsystem, and *Decryption* must be replaced by *Decryption(new)* for every receiver's subsystem. This upgrade is also a dependent update. The state (the receiver's public keys) must be transferred from *Encryption* into *Encryption(new)*. In particular, when the session key is required to never be repeated, *Encryption* needs a critical application state which prevents it from repeating the session keys already produced. The state (the receiver's private key) must be transferred from *Decryption* into *Decryption(new)*.

The algorithm for message digesting is to be upgraded. Therefore, *Message Digest* must be replaced by *Message Digest(new)* for both the sender's subsystem and every receiver's subsystem. This upgrade is a dependent update as well.

The algorithm for signing (digest encryption) is to be upgraded. Therefore, *Digest Encryption* must be replaced by *Digest Encryption(new)* for the sender's subsystem, and *Signature Decryption* must be replaced by *Signature Decryption(new)* for every receiver's subsystem. This upgrade is also a dependent update. The state (the sender's private keys) must be transferred from *Digest Encryption* into *Digest Encryption(new)*, and the state (the sender's public keys) must be transferred from *Signature Decryption* into *Signature Decryption(new)*.

The above reconfiguration requirements also result in the replacement of *Sender* by *Sender(new)* and the replacement of *Receiver* by *Receiver(new)*. The reason is that the interfaces of these components must be upgraded to accommodate the new required-service *compression* and *decompression*, respectively. *Signing*, *Delivery*, *Reception*, and *Signature Verification* are *shared* components because the same provided services are invoked by both the old and the new subsystem. *Signing* and *Signature Verification* have *bi-links* (illustrated in Fig. 7) because the provided services they need are being upgraded. The bi-links in the reconfiguration scenario require dynamic version management if coexistence is to be supported. Reconfiguration with safe coexistence must ensure totally independent workflow between the two invocation chains (illustrated in Fig. 8) because the two chains have shared components (*Signing* and *Signature Verification*) that make further invocations on unshared components.

5.3 QoS Testing Procedure

To evaluate the reconfiguration strategies in Table 2 and compare related work, a general, parameterized, and reproducible testing environment is necessary. In addition, appropriate evaluation criteria are important to effectively reflect the actual impact of reconfiguration on the system's QoS.

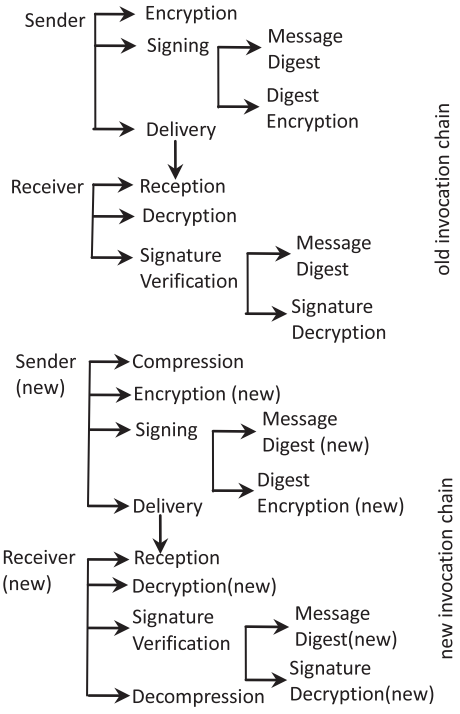

Fig. 8. The invocation chains in the original and target DEDSS. During the coexistence period, the two chains share components *Delivery*, Reception, *Signing*, and *Signature Verification*, of which the last two components further invoke unshared components.

5.3.1 Testing Environment

In RCM, the application system's configuration is specified by the RCM specification language (in XML schema [22]). For DEDSS configuration, the number of receivers is user specified to reflect the problem scale. The *concentrate* (this term is used in RCM) of a *component* reflects the functionally computing load of the component; the concentrate of *initializing* or *removing* a component reflects the computing and communication load of the initialization or removal of the component; the concentrate of *state transfer* reflects the computing load if the transfer is performed by a deep copy of state variables. All types of concentrate are parameterized for high-load operations, and their values are quantified as user-specified time delays in ms. When the application logic of DEDSS is a secured data transmission system, the computing and communication load of the system is not fixed to a particular application algorithm or communication protocol but parameterized for extensive simulations. For example, this parameterization can be used to test the scalability of RCM in support of reconfiguration of systems with different sizes. However, the concentrate of the *virtually instantaneous* operations is not user specified. Its actual computing load is CPU instruction determined and used to check whether they are really made instantaneous as expected in Section 4.2.2 or have significant impacts on QoS.

To test a strategy for different workloads, the number of concurrent application threads, the number of invocations to the system services from each thread, and the time delay to start the next thread or reinvoke a service for a given thread are all user-specified. These parameters are designed to create varying workloads, especially CPU saturation versus nonsaturation cases on different hardware or software

platforms. The time interval to gather statistics on the system throughput and the frequency at which to sample the system responsiveness are also user specified. For example, a parameter selection might specify that throughput be sampled every 4 s, and response time checked for every request to be processed by the system. These parameters are specified by the user through *test agents* via RCM APIs.

To enable sound evaluation, the test context should be independent so that the test results can be reproducible and validated on different platforms. One step toward platform independence is that RCM is developed using Java, which enables it run on any machine supporting a Java Virtual Machine (JVM). However, tests show that Java threading policy on a multiple-processor machine is different from that on a single-processor machine, even for the same operating system version. Consequently, the challenge is to design independent reconfiguration strategies (as described in Section 5.1.3).

Testing was conducted on two machines, one a single-processor machine, the other a two-processor machine. Both machines were configured with the same version of Microsoft Windows XP operating system and the same Java SE 6 runtime. The different platforms were chosen to demonstrate the independence of RCM reconfiguration strategies and the reproducibility of results.

On *machine-1* (Intel Core 2 Duo, CPU 4300 @ 1.80 GHz and 1.79 GHz, and 2 GB RAM), the testing environment was created for CPU saturation, while on *machine-2* (Intel Pentium 4, CPU @ 3.00 GHz and 1 GB RAM), the environment was created for CPU nonsaturation. The reconfiguration strategies in Table 2 were tested on both machines. In particular, for CPU saturation, strategies *qos-ts0.55* and *qos-ts0.35* were applied while measuring system response time. These strategies use the same 100 ms time-slice but 55 ms or 35 ms runnable time slots differently for running the reconfiguration thread. The strategies *qos-ts0.45* and *qos-ts0.3* (1,000 ms time slice with 450 ms or 300 ms runnable time-slots for running the reconfiguration thread) were tested for the system throughput. The difference of these parameters aimed to show the reproducibility of results, and the difference of QoS values aimed to demonstrate the controllability of the time-slice scheduling on overheads. This is achievable because the reconfiguration thread is restricted differently by these parameters.

5.3.2 Evaluation Criteria

Hillman and Warren [14] used Warren's [35] algorithm for the reconfiguration of a simple Electronic Point of Sale (EPOS) application. They used total reconfiguration time, the block time of each component, and synchronization time to achieve system quiescence to evaluate the performance of the algorithm. Hillman and Warren [15] went a step further in performance analysis in order to aid understanding of reconfiguration. They proposed three criteria: 1) global consistency, 2) application contributions, and 3) performance for the evaluation. The performance was measured by reconfiguration time and disturbance to the application. In Hillman and Warren [15], two reconfiguration algorithms were compared on the OpenRec [14] framework in terms of the *wait* time that the component spent on waiting for data from its upstream updating components. The first algorithm was Mitchell et al.'s [24] algorithm and the second algorithm was Warren [35]. They demonstrated that the former was better in maintaining the performance of the running system (in the study case of the reconfiguration of a simple component-based router) because the former exploited the coexistence feature.

In Bidan et al. [6], reconfiguration time was used to evaluate the overheads of the proposed reconfiguration algorithm on a simple client/server application. In Truyen et al. [33], reconfiguration time was reported for adding, removing, and replacing fragmentation services for the Instant Messaging Service (IMS) study case. In Ajmani et al. [2], the proposed method was evaluated on the prototype framework: Upstart. The overheads imposed by the upgrade layer were tested with 100,000 null RPC loopback and crossover calls, respectively, and 100 Transport Control Protocol (TCP) transfers of 100 MB data. The absolute time of these operations was compared between the baseline (without upgrade layer) performance and the Upstart performance.

Similar evaluations of reconfiguration overhead by reconfiguration execution time are recently reported in Leger et al. [21], where reconfiguration overhead was compared for reconfigurations without and with Java Remote Method Invocation (RMI) transactions, and Surajbali et al. [31], where reconfiguration overhead was compared for reconfigurations without and with Consistency Framework (COF) under different consistency constraints in aspect-oriented context, such as system consistency aspects and compositional aspects.

The above metrics are not directly user-observable. We cannot see how the system's QoS is in terms of these metrics from a user's point of view. After all, end users experience the system's QoS, and the framework maintains QoS for end users. Consequently, if these metrics cannot be observed by end users, they are too indirect to be appropriate for QoS evaluation. In addition, it is questionable whether reconfiguration time is always important for evaluation. Suppose that a reconfiguration lasts a very short period but causes very sudden and sharp deterioration on QoS, and another reconfiguration lasts for significantly longer but has little impact on QoS. When a malfunction must be rectified quickly, the former is better and sometimes shutting down the system is necessary. However, when QoS is a major concern, the latter is more likely to be acceptable for system upgrading.

Given the impact of reconfiguration, the common QoS values: *system throughput* and *response time* should be the key metrics to evaluate reconfiguration impacts; they are the direct experiences of end users. In the tests, we restrict RCM to transmit the same size data packages, and therefore these two metrics are defined precisely as follows: *system throughput* is the number of requests that the system can process in a predetermined time interval and *system response time* is the time delay between request submission and until its completion as confirmed by its response.

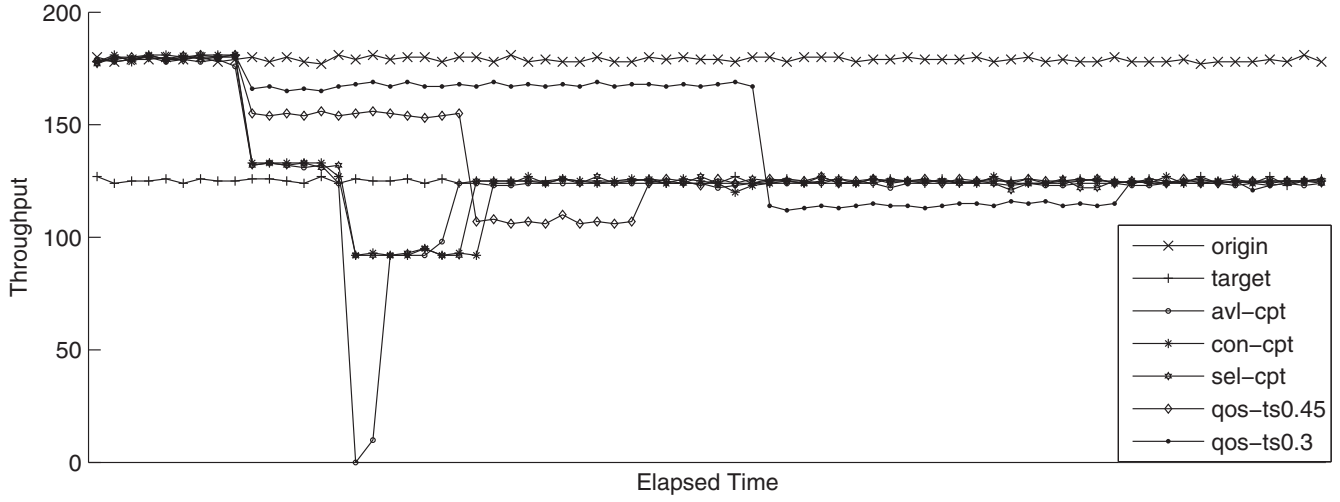

Fig. 9. The system throughput under different reconfiguration strategies is compared for CPU saturation using the original and target system's throughput as the comparison baselines.

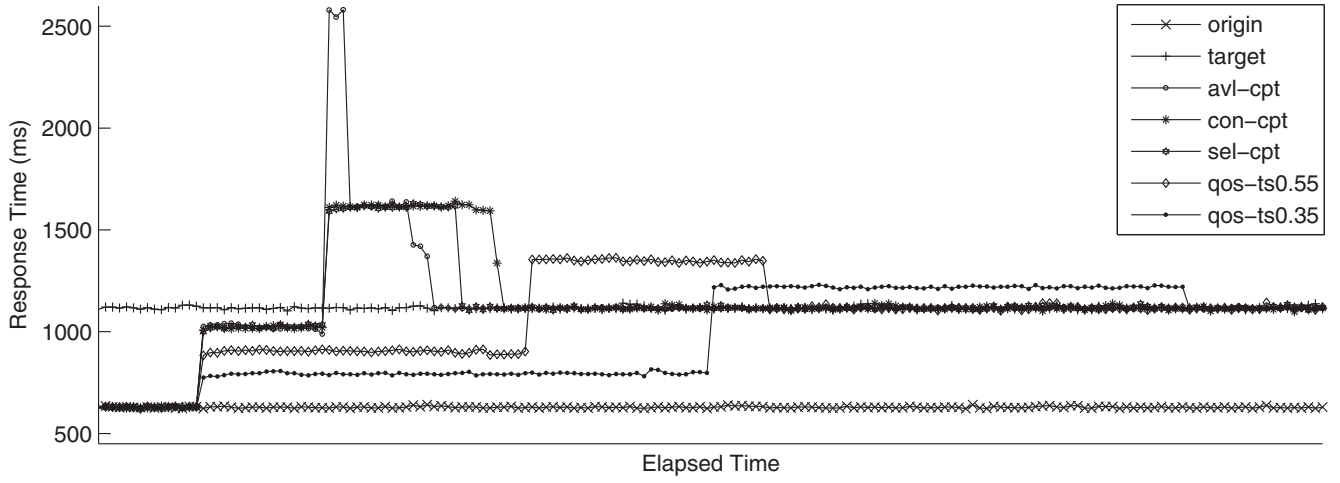

Fig. 10. The system response time (ms) under different reconfiguration strategies is compared for CPU saturation using the original and target system's response time as the comparison baselines.

5.4 Results and Analysis

Results are depicted using line charts and analyzed for each strategy. Evaluation of reconfiguration strategies on QoS impact follows.

5.4.1 Testing Results

The presence of reconfiguration must not compromise application consistency. Reconfiguration correctness is application dependent. In DEDSS, reconfiguration correctness is evaluated using two criteria. We suppose that there is no malicious interference with experiments; therefore, 1) the independent workflow can be validated if all data items are successfully decrypted and all signatures are successfully verified. Otherwise, there must be transaction interleaving and unsuccessful data verification. 2) The completion of transactions can be validated if the number of submitted requests matches the number of completed requests. Otherwise, there must have been failed transactions during reconfiguration.

We conducted about 50 tests. In all tests, successful data decryption, signature verification, and the number match always occurred on both machines.

To make performance comparison clearer, the QoS values are shown for the old (denoted as *origin*) and the new (denoted as *target*) subsystem as comparison baselines when there is no reconfiguration involved. QoS values have been recorded for each reconfiguration strategy so that how QoS was affected by that strategy can be compared with other strategies. The system's throughput and response time under different reconfiguration strategies are reported in Figs. 9 and 10, respectively, for CPU saturation and in Figs. 11 and 12 for CPU nonsaturation, where the horizontal axis represents elapsed time during a reconfiguration.

5.4.2 Result Analysis

In all tests both for CPU saturation and nonsaturation cases, there was no time interval for the *transformation* phase to be identified for *qos-ts* (CPU saturation) or *qos-pre* (CPU nonsaturation). This demonstrated that operations in this phase were really made virtually instantaneous, and the scheduling polices in *qos-ts* or *qos-pre* maintained this feature throughout the whole transformation phase.

In all the line charts, the biggest drop of throughput and responsiveness of *avl-cpt* came from blocking operations

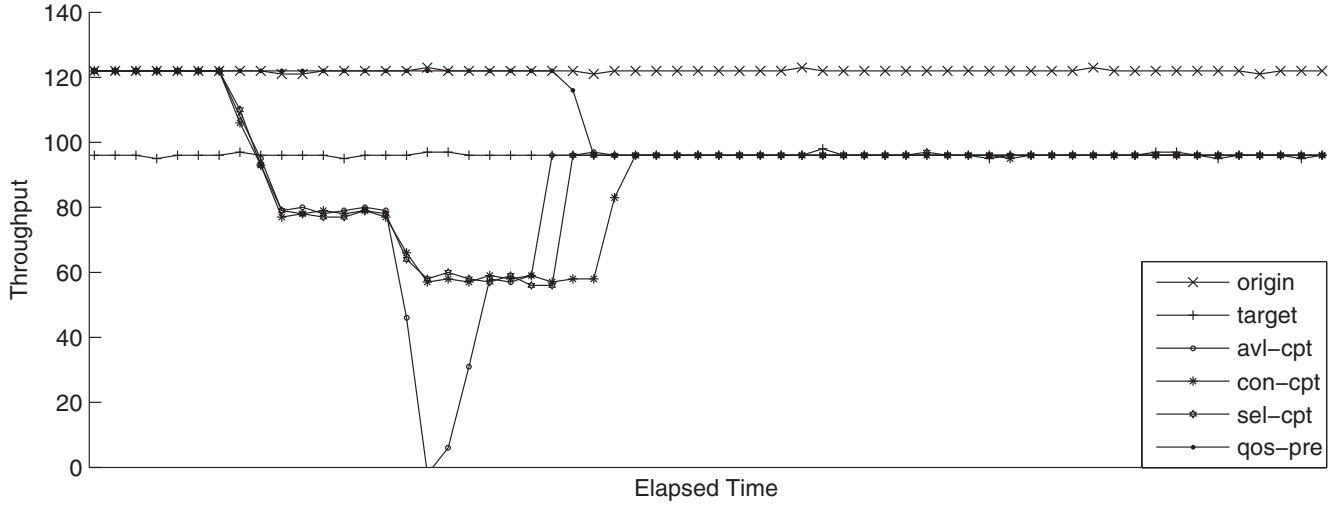

Fig. 11. The system throughput under different reconfiguration strategies is compared for CPU nonsaturation using the original and target system's throughput as the comparison baselines.

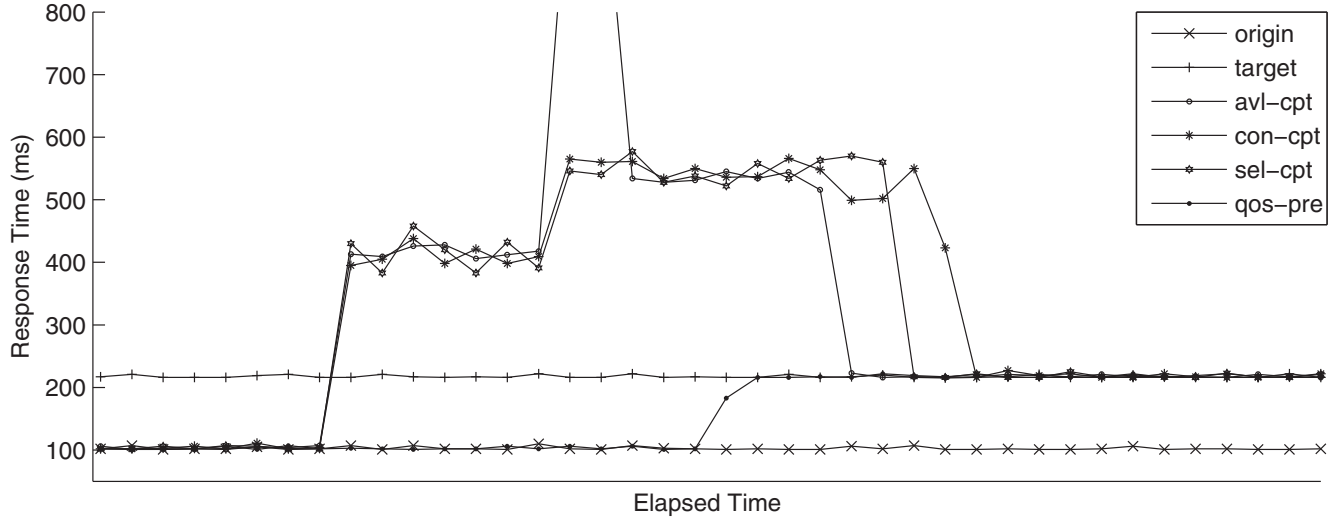

Fig. 12. The system response time (ms) under different reconfiguration strategies is compared for CPU nonsaturation using the original and target system's response time as the comparison baselines.

(for quiescence), which, however, were removed from other strategies and therefore their performance improved. A similar result on service disruption from blocking operations was reported by Truyen et al. [33]. The most recently quantitative reports of the impact of quiescence on running systems include Guay et al. [11] and Wurthinger et al. [38], of which the former showed a big drop in throughput and the latter showed a significant increase in response time. Their results comply with the performance of strategy: *avl-cpt* reported here.

The performance of *sle-cpt* was almost the same as that of *con-cpt* except that the drop in throughput and responsiveness for *sle-cpt* was shorter than for *con-cpt*. This better performance of *sle-cpt* than *con-cpt* is due to the fact that the former was stateless equivalent (virtually instantaneous for state transfer), but the latter needed a deep copy of states variables for state transfer and thereby consumed considerable time and blocked the workflow.

With reference to CPU saturation, the system throughput (Fig. 9) declined and response time (Fig. 10) increased for all the reconfiguration strategies, including the QoS assurance strategy *qos-ts*. The reasons were: 1) CPUs were always saturated by ongoing transactions; and 2) intensive reconfiguration operations consumed a certain amount of resources on which ongoing transactions relied for maintaining the usual throughput or responsiveness. However, a significant difference between *qos-ts* and other strategies was that the overhead was controllable for *qos-ts*. The controllability can be clearly observed in the better performance of *qos-ts0.3* over *qos-ts0.45* (throughput in Fig. 9) and *qos-ts0.35* over *qos-ts0.55* (response time in Fig. 10). Note that the former allows a smaller runnable time slot to the reconfiguration thread. Therefore, reconfiguration was restricted more by resource competition, and ongoing transactions were guaranteed resources needed for maintaining QoS.

With reference to CPU nonsaturation, the strategy *qos-pre* exhibited no impact on the system performance. The system throughput (Fig. 11) and response time (Fig. 12) for *qos-pre* changed directly from the original to the target performance line. The reason was that reconfiguration performed by *qos-pre* only made use of CPU free time and never caused

suspension of ongoing transactions. However, throughput decreased and response time increased for all other strategies because they all had the same execution priority as that of ongoing transactions and competed for resources. That resulted in suspension of ongoing transactions and therefore performance declined.

In addition, there is a notable positive correlation between throughput and response time for the same reconfiguration strategy in all tests. This further confirms that results are reliable and reproducible.

In conclusion, the system's downtime can be avoided by *avl-cpt*(availability), but the system experiences sudden and significant deterioration on QoS, which is intolerable to some multimedia applications like these in Mitchell et al. [24]. With continuity, *con-cpt* exhibits a better QoS performance than *avl-cpt*. Furthermore, with stateless equivalence, *sel-cpt* exhibits a better QoS performance than *con-cpt*. The best solutions to QoS assurance among all the strategies are *qos-ts* and *qos-pre*, which fully control reconfiguration impact on QoS. Consequently, QoS strength can be sorted in the ascending order: *avl-cpt*(availability), *con-cpt*(continuity), *sel-cpt*(continuity plus stateless equivalence), and *qos-ts*(QoS assurance) or *qos-pre*(QoS assurance). The classified QoS characteristics can be fully achieved under the acceptable state-sharing constraints (Section 4.2.1).

6 Conclusion and Future Work

This paper has comprehensively addressed the problem of QoS assurance for dynamic reconfiguration of component-based software systems. The research underpinning this paper has successfully attempted QoS assurance from four aspects:

1. The coexistence property has been identified as a necessary condition for QoS assurance because it logically removes blocking operations. DVM has been attempted as an effective mechanism to preserve application consistency and availability in the face of coexistence of the old and new subsystem.
2. State transfer has been identified as an invisible factor which could cause significant impact on QoS. Stateless equivalence has been attempted by applying state sharing under the required constraints and timing control.
3. Controllability of resource competition during reconfiguration has been delegated to different scheduling policies, which are applied to different reconfiguration phases under different resource usages.
4. Prior work has been realized by particular reconfiguration strategies, which have been evaluated on the proof-of-concept prototype RCM. System performance under different strategies has been evaluated quantitatively. The better solutions to QoS assurance have been experimentally confirmed to come from RCM.

There exist some open issues in the QoS assurance model RCM which constitute future work. First-in-first-out order is important to some multimedia applications. RCM has no control over the order of request processing during coexistence and delegates this issue to the application layer. To apply overhead control, adaptive scheduling should be investigated for the case when system resource usage varies between saturation and nonsaturation. The applicability of state-sharing policy is worth further investigation, and relaxation of the constraints on state sharing is an issue for further research. To make the proposed QoS assurance framework more applicable to distributed environments, component state migration will be the biggest challenge. It is expected that remote referencing will enable initial control of QoS for state migration. However, appropriate timing needs to be researched for physical migration of the state for QoS assurance.

Acknowledgments

This work was partially supported by Central Queensland University Australia under Research Advancement Awards Scheme (RAAS) Grants, 2006-2007. The author would like to thank Dr. Ian Peake from RMIT University Australia for his proofreading of this paper.

References

[1] S. Ajmani, "A Review of Software Upgrade Techniques for Distributed Systems," http://pmg.csail.mit.edu/~ajmani/papers/review.pdf, 2004.
[2] S. Ajmani, B. Liskov, and L. Shrira, "Modular Software Upgrades for Distributed Systems," *Proc. 20th European Conf. Object-Oriented Programming*, pp. 452-476, 2006.
[3] J.P.A. Almeida, M. van Sinderen, L.F. Pires, and M. Wegdam, "Handling QoS in MDA: A Discussion on Availability and Dynamic Reconfiguration," *Proc. Workshop Model Driven Architecture: Foundations and Application*, pp. 91-96, 2003.
[4] J.P.A. Almeidaa, M. van Sinderena, L.F. Piresa, and M. Wegdama, "Platform-Independent Dynamic Reconfiguration of Distributed Applications," *Proc. 10th IEEE Int'l Workshop Future Trends of Distributed Computing Systems*, pp. 286-291, 2004.
[5] N. Arshad, D. Heimbigner, and A.L. Wolf, "Deployment and Dynamic Reconfiguration Planning for Distributed Software Systems," *Proc. 15th IEEE Int'l Conf. Tools with Artificial Intelligence*, pp.39-46, 2003.
[6] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras, "A Dynamic Reconfiguration Service for CORBA," *Proc. Fourth Int'l Conf. Configurable Distributed Systems*, pp. 35-42, 1998.
[7] J.E. Cook and J.A. Dage, "Highly Reliable Upgrading of Components," *Proc. Int'l Conf. Software Eng.*, pp. 203-212, 1999.
[8] J. Dowling and V. Cahill, "The K-Component Architecture Meta-Model for Self-Adaptive Software," *Proc. Third Int'l Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, pp. 81-88, 2001.
[9] J. Dowling and V. Cahill, "Dynamic Software Evolution and the K-Component Model," *Proc. OOPSLA 2001 Workshop Software Evolution*, 2001.
[10] H. Evans, "DRASTIC and GRUMPS: The Design and Implementation of Two Run-Time Evolution Frameworks," *IEE Proc. Software*, vol. 151, no. 2, pp. 30-48, Apr. 2004.
[11] W.L. Guay, S.A. Reinemo, B.D. Johnson, and L. Holen, "Host Side Dynamic Reconfiguration with InfiniBand," *Proc. Int'l Conf. Cluster Computing*, pp. 126-135, 2010.
[12] J. Gorinsek, S. Van Baelen, Y. Berbers, and K. De Vlaminck, "Managing Quality of Service during Evolution Using Component Contract," *Proc. Second Int'l Workshop Unanticipated Software Evolution*, pp. 57-62, 2003.
[13] M. Hicks, J.T. Moore, and S. Nettles, "Dynamic Software Updating," *ACM Trans. Programming Languages and Systems*, vol. 27, no. 6, pp. 1049-1096, 2005.
[14] J. Hillman and I. Warren, "An Open Framework for Dynamic Reconfiguration," *Proc. 26th Int'l Conf. Software Eng.*, pp. 594-603, 2004.
[15] J. Hillman and I. Warren, "Quantitative Analysis of Dynamic Reconfiguration Algorithms," *Proc. Int'l Conf. Design, Analysis and Simulation of Distributed Systems*, 2004.
[16] D. Jackson, "Alloy: A Lightweight Object Modeling Notation," *ACM Trans. Software Eng. and Methodology*, vol. 11, no. 2, pp. 256-290, 2002.

[17] N. Janssens, L. Desmet, S. Michiels, and P. Verbaeten, "NeCoMan: Middleware for Safe Distributed Service Deployment in Programmable Networks," *Proc. Workshop Adaptive and Reflective Middleware,* pp. 256-261, 2004.

[18] N. Janssens, E. Truyen, F. Sanen, and W. Joosen, "Adding Dynamic Reconfiguration Support to JBoss AOP," *Proc. First Workshop Middleware-Application Interaction,* pp. 1-8, 2007.

[19] D.K. Kim and S. Bohner, "Dynamic Reconfiguration for Java Applications Using AOP," *Proc. IEEE Southeastcon,* pp. 210-215, 2008.

[20] 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.

[21] M. Leger, T. Ledoux, and T. Coupaye, "Reliable Dynamic Reconfigurations in a Reflective Component Model," *Proc. Int'l Symp. Component-Based Software Eng.,* pp. 74-92, 2010.

[22] W. Li, "DynaQoS-RDF: A Best Effort for QoS Assurance of Dynamic Reconfiguration of Dataflow Systems," *J. Software Maintenance and Evolution: Research and Practice,* vol. 21, no. 1, pp. 19-48, 2009.

[23] W. Li, "Evaluating the Impacts of Dynamic Reconfiguration on the QoS of Running Systems," *J. Systems and Software,* vol. 84, pp. 2123-2138, 2011.

[24] S. Mitchell, H. Naguib, G. Coulouris, and T. Kindberg, "A QoS Support Framework for Dynamically Reconfigurable Multimedia Applications," *Proc. Int'l Conf. Distributed Applications and Interoperable Systems,* pp. 17-30, 1999.

[25] L.E. Moser, P.M. Melliar-Smith, and L.A. Tewksbury, "Online Upgrades Become Standard," *Proc. 26th Ann. Int'l Computer Software and Applications Conf.,* pp. 982-988, 2002.

[26] N. De Palma, P. Laumay, and L. Bellissard, "Ensuring Dynamic Reconfiguration Consistency," *Proc. Sixth Int'l Workshop Component-Oriented Programming,* 2001.

[27] A. Rasche and A. Polze, "ReDAC—Dynamic Reconfiguration of Distributed Component-Based Applications with Cyclic Dependencies," *Proc. 11th IEEE Int'l Symp. Object Oriented Real-Time Distributed Computing,* pp. 322-330, 2008.

[28] T. Senivongse, "Enabling Flexible Cross-Version Interoperability for Distributed Services," *Proc. Int'l Symp. Distributed Objects and Applications,* pp. 201-210, 1999.

[29] M. Solarski and H. Meling, "Towards Upgrading Actively Replicated Servers On-the-Fly," *Proc. 26th Ann. Int'l Computer Software and Applications Conf.,* pp. 1038-1043, 2002.

[30] J.G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and L. Neamtiu, "Mutatis Mutandis: Safe and Predictable Dynamic Software Updating," *ACM Trans. Programming Language and Systems,* vol. 29, no. 4, pp. 183-194, 2007.

[31] B. Surajbali, P. Grace, and G. Coulson, "Preserving Dynamic Reconfiguration Consistency in Aspect Oriented Middleware," *Proc. Ninth Workshop Aspects, Components, and Patterns for Infrastructure Software,* pp. 33-40, 2010.

[32] L.A. Tewksbury, L.E. Moser, and P.M. Melliar-Smith, "Live Upgrades of CORBA Applications Using Object Replication," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 488-497, 2001.

[33] E. Truyen, N. Janssens, F. Sanen, and W. Joosen, "Support for Distributed Adaptations in Aspect-Oriented Middleware," *Proc. Seventh Int'l Conf. Aspect-Oriented Software Development,* pp. 120-131, 2008.

[34] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquillity: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Trans. Software Eng.,* vol. 33, no. 12, pp. 856-868, Dec. 2007.

[35] I. Warren, "A Model for Dynamic Configuration Which Preserves Application Integrity," PhD thesis, Lancaster Univ., 2000.

[36] I. Warren, J. Sun, S. Krishnamohan, and T. Weerasinghe, "An Automated Formal Approach to Managing Dynamic Reconfiguration," *Proc. 21st IEEE Int'l Conf. Automated Software Eng.,* pp. 37-46, 2006.

[37] T. Wurthinger, W. Binder, D. Ansaloni, P. Moret, and H. Mossenbock, "Improving Aspect-Oriented Programming with Dynamic Code Evolution in an Enhanced Java Virtual Machine," *Proc. Seventh Workshop Reflection, AOP and Meta-Data for Software Evolution,* 2010.

[38] T. Wurthinger, C. Wimmer, and L. Stadler, "Dynamic Code Evolution for Java," *Proc. Eighth Int'l Conf. Principles and Practice of Programming in Java,* pp. 10-19, 2010.

[39] J. Zhang, Z. Yang, B. Cheng, and P. McKinley, "Adding Safeness to Dynamic Adaptation Techniques," *Proc. ICSE 2004 Workshop Architecting Dependable Systems,* pp. 17-21, 2004.

Wei Li received the BS, MS, and PhD degrees, all in computer science, from Harbin University of Science and Technology, China, Harbin Institute of Technology, China, and the Institute of Computing Technology of the Chinese Academy of Sciences, China, in 1986, 1989, and 1998, respectively. He is currently a senior lecturer in information technology with the School of Information & Communication Technology, Central Queensland University, Australia. He has been a reviewer for a number of international journals and a program committee member for 30 international conferences in his research domain. His research interests include dynamic software architecture, P2P volunteer computing, and multi-agent systems.