The Journal of Systems and Software 84 (2011) 2123–2138

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Evaluating the impacts of dynamic reconfiguration on the QoS of running systems

Wei Li*

Centre for Intelligent and Networked Systems, and School of Information & Communication Technology, Central Queensland University, Rockhampton, QLD 4702, Australia

ARTICLE INFO

Article history:
Received 27 June 2010
Received in revised form 18 May 2011
Accepted 25 May 2011
Available online 12 July 2011

Keywords:
Dynamic reconfiguration
QoS assurance
Software maintenance
Software evolution
Quantitative analysis

ABSTRACT

A major challenge in dynamic reconfiguration of a running system is to understand in advance the impact on the system's Quality of Service (QoS). For some systems, any unexpected change to QoS is unacceptable. In others, the possibility of dissatisfaction increases due to the impaired performance of the running system or unpredictable errors in the resulting system. In general it is difficult to choose a reasonable reconfiguration approach to satisfy a particular domain application. Our investigation on this issue for dynamic approaches is four-fold. First, we define a set of QoS characteristics to identify the evaluation criteria. Second, we design a set of abstract reconfiguration strategies bringing existing and new approaches into a unified evaluation context. Third, we design a reconfiguration benchmark to expose a rich set of QoS problems. Finally, we test the reconfiguration strategies against the benchmark and evaluate the test results. The analysis of acquired results helps to understand dynamic reconfiguration approaches in terms of their impact on the QoS of running systems and possible enhancements for newer QoS capability.

1. Introduction

Dynamic reconfiguration is a widely studied topic in the runtime evolution of software systems. One way to assist appropriate use of a particular dynamic reconfiguration approach is to understand the impact that the approach may have on a running system. This article conducts an evaluation with respect to *Quality of Service* (QoS), by which we refer to end-user observable performance attributes such as *throughput* and *response time* of a running system, and *QoS assurance*, by which we refer to the capability of a reconfiguration approach to maintain pre-determined QoS levels for a running system under reconfiguration (RSUR). The background and necessity of this research are as follows.

It is necessary for a software system to evolve continuously in order to remain useful (Kramer & Magee, 1990; Soria et al., 2009; Vandewoude et al., 2007). This evolution adapts the system to the changing business needs that diverge from the original requirements placed upon it (Morrison et al., 2007). The evolution comprises incremental upgrade of the functionality and modification of the topology of a software system (Fung & Low, 2009). In its lifecycle, a software system may evolve multiple times; *a reconfiguration* refers specifically to a specific process of reconfiguration at a specific point in the history of a system evolution.

The evolution of software systems can be accomplished by static or dynamic reconfiguration. *Static reconfiguration* involves stopping a running system, recompiling its binaries, rebuilding its data, altering its topology and then restarting the system. A static reconfiguration is performed out of band and is therefore not viable for the applications that are featured as mission-critical or uninterruptible (Vandewoude et al., 2007). For mission-critical services or 24/7 business services, such as those described by Warren et al. (2006), a running system cannot be reconfigured statically due to unacceptable down time. For other systems, down time reduces the productivity of users and increases costs for the businesses (Patterson, 2002).

Dynamic reconfiguration involves upgrading/altering a system's functionality and topology at runtime via addition, deletion and replacement of components and connections. Dynamic reconfiguration promotes service availability and self-adaptation because it is able to make reconfiguring or adaptive steps as a part of normal system execution.

From the early work of Kramer and Magee (1990) to the recent work of Morrison et al. (2007), Soria et al. (2008, 2009) and Vandewoude et al. (2007), dynamic reconfiguration has advanced significantly in modeling (Ajmani et al., 2006; Almeida et al., 2004; Dowling & Cahill, 2001a,b; Warren et al., 2006) and implementation (Evans, 2004; Janssens et al., 2007; Kim & Bohner, 2008; Rasche & Polze, 2008; Tewksbury et al., 2001; Truyen et al., 2008).

However, recently Vandewoude et al. (2007) re-investigated a typical dynamic approach (Kramer & Magee, 1990) and noted that it could result in significant disruption to the application being updated. That work raised the issue that impact evaluation is significant for dynamic reconfiguration.

Moreover, existing approaches have been evaluated individually, but not quantitatively compared in a unified evaluation context. As a result, the questions: *how is dynamic reconfiguration*

* Tel.: +61 7 4930 9686; fax: +61 7 4930 9729.
E-mail address: w.li@cqu.edu.au

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.05.060

different from static reconfiguration, and *how are dynamic approaches different from each other* remain open.

In this article we argue that addressing dynamic reconfiguration on a running system without systematic evaluation increases the possibility of user dissatisfaction, due to impaired performance or unpredictable errors. Furthermore, without comparison, it is difficult for a developer to choose the right dynamic approach to satisfy a particular domain application or enhance an available approach for a new use.

This article argues that evaluation of dynamic reconfiguration is a *quantitative* rather than simply *qualitative* issue as described as follows: (1) keeping a system operational under reconfiguration (Dowling & Cahill, 2001a,b; Kramer & Magee, 1990); (2) supporting changes to an application's structure and functionality at runtime without shutting it down (Fung & Low, 2009); and (3) integrating new components, or modifying or removing existing ones while a system is running (Soria et al., 2009). The non-stop feature (availability) is just a qualitative aspect and far from an appropriate ground for successful dynamic reconfiguration. The premise that dynamic reconfiguration is more suitable for software evolution lies in its promotion for the QoS assurance of a RSUR. Otherwise static reconfiguration is more acceptable in terms of simplicity.

The limitations of the state-of-the-art as compared with a systematic evaluation are:

1. there is no set of unified criteria to evaluate the impact of dynamic reconfiguration on the QoS of a RSUR;
2. there is no evaluation benchmark covering a rich set of available QoS scenarios;
3. there is no abstraction or representation for modeling available reconfiguration approaches in a unified evaluation context;
4. there is no evaluation on the QoS of a RSUR with respect to the common QoS metrics: throughput and response time;
5. there is no identification or comparison of the QoS assurance capabilities offered by alternative approaches.

Reconfiguration approaches have been evaluated with respect to isolated criteria (as reviewed in the next section), but they have not been realized into a unified evaluation context. Therefore, little quantitative comparison in terms of impact on the QoS of RSURs between these approaches can be found in the current literature.

This article addresses these gaps by proposing an evaluation method and an evaluation benchmark, and conducting a set of experimental evaluations. The stimulus for this article is prior work (Li, 2009) which explores QoS assurance for dynamic reconfiguration of dataflow systems. The novelty of this article with respect to the prior work lies in that it develops the original evaluation ideas into a more systematic evaluation. Thus the novel contributions of this article are to:

1. propose fundamental QoS characteristics as the criteria to evaluate QoS assurance capabilities of available dynamic reconfiguration approaches;
2. propose criteria for discovering the QoS assurance capability of a given approach;
3. propose a representation model to model abstractly available dynamic approaches so that they can be evaluated and compared on a unified evaluation context;
4. introduce a more complex (therefore more realistic) reconfiguration scenario to accommodate a richer set of reconfiguration problems;
5. design an evaluation benchmark to expose existing QoS problems and facilitate the quantitative measurement of QoS in terms of throughput and responsiveness;
6. evaluate and compare the QoS assurance capabilities of existing dynamic approaches.

The remainder of this article is structured as follows. A detailed review of related work in particular with respect to evaluation of dynamic reconfiguration approaches is presented in Section 2. A set of evaluation criteria for QoS assurance capabilities is proposed in Section 3. A qualitative classification of available reconfiguration approaches is presented in Section 4. A set of criteria for discovering QoS assurance capabilities of existing reconfiguration approaches is proposed in Section 5. An evaluation benchmark and its two architectural prototypes are presented in Section 6. New reconfiguration strategies are proposed as abstraction and realization of available approaches in Section 7. The reconfiguration strategies are evaluated and compared for their QoS assurance capabilities in Section 8. Section 9 concludes by outlining the QoS evaluation results arising out of this research.

2. Related work

Related work varies with respect to the kind that dynamic reconfiguration approaches are evaluated.

2.1. Individual evaluation

Some dynamic approaches have been evaluated only with respect to local, isolated criteria. In Bidan et al. (1998) a simple client/server application was used as a case study to evaluate the proposed reconfiguration algorithm for its overhead as measured by reconfiguration time. The case study has limitations making it unsuitable for more inclusion in a more extensive evaluation, due to a lack of complexity, flexibility, and coverage of existing QoS problems. Ajmani et al. (2006) evaluated the reconfiguration framework *Upstart*. To evaluate the overhead imposed by a so-called upgrade layer dedicated to dynamic reconfiguration they tested 100,000 null RPC (Remote Procedure Call) loopback calls and crossover calls; 100 TCP transfers of 100MB data were conducted. The time consumed by the operations was used to compare the performance between the base framework (without upgrade layer) and Upstart. The results showed that the upgrade layer introduced only modest overhead. Truyen et al. (2008) reported the reconfiguration time for adding, removing and replacing a fragmentation service for their case study, an Instant Messaging Service. The overhead of introducing a reconfiguration proxy was reported to be insignificant, but it is not clear to what extent this result generalizes.

The most recent evaluations of reconfiguration overhead include Leger et al. (2010) and Surajbali et al. (2010). Both reported overhead at reconfiguration time. The former compared reconfiguration with and without Java RMI (Remote Method Invocation) transactions, and the latter compared reconfiguration with and without COF (Consistency Framework).

Two important works in dynamic reconfiguration are *quiescence* (Kramer & Magee, 1990) and *tranquility* (Vandewoude et al., 2007). Quiescence refers to a system state where a node (software component) is not involved in any transactions and will neither receive nor initiate new transactions. Quiescence is proved by Kramer and Magee (1990) to be a sufficient condition for preservation of application consistency during dynamic updates. Improving on quiescence, tranquility exploits the notion of a minimal passivated set of components for reducing disruption on the ongoing transactions. Although Vandewoude et al. theoretically analyzed that tranquility has less disruption than quiescence, they did not conduct any quantitative comparisons between tranquility and quiescence in terms of disruption to the QoS of a RSUR. This is evidenced by the following. In Vandewoude et al. (2007), a tranquility-based approach was evaluated for a component-based web shop. Evaluation involved applying system loads of 10, 50, and 100 concurrent clients, each sending one request per second with random pauses of 100 ms. The operation time and the number of checked messages before reaching a tranquil state were observed. In all 40 executions of the evaluation scenario, tranquility occurred. Furthermore, Vandewoude et al. reported the number of messages that had been checked before a tranquil state was reached for another 200 experiments with 10 concurrent clients. The results were normally distributed.

2.2. *Algorithm comparison*

Comparison of reconfiguration algorithms has appeared in a few of articles. Hillman and Warren (2004a) developed the OpenRec framework, which can accommodate, analyze and compare different reconfiguration algorithms. They tested Warren's (2000) algorithm for the reconfiguration of a simple EPOS (Electronic Point of Sale) application with respect to total reconfiguration time, blocking time of each component, and time to quiescence. In Hillman and Warren (2004b), they went a step further in performance analysis in order to aid understanding of reconfiguration impacts. They proposed three criteria: (1) global consistency; (2) application contributions; and (3) performance. Performance was measured by reconfiguration time and disruption to the applications. They argued that it would be beneficial to be able to observe the performance of an algorithm for a given application scenario because an informed decision could then be made in choosing an adequate algorithm for reconfiguration. Based on this motivation, they compared two reconfiguration algorithms on OpenRec framework in terms of the *wait-time* that the components spent on waiting for data from its upstream updating components for a component-based router. The first algorithm was that of Mitchell et al. (1999), which is applicable to the reconfiguration of systems with pipe-and-filter structure. The algorithm is logically non-blocking because it exploits the coexistence of the new and old sub-system. The second algorithm, that of Warren (2000), needs to apply quiescence for preserving application consistency. By comparing wait-time, they demonstrated that the former was better in maintaining QoS of the RSUR.

Truyen et al. (2008) compared two algorithms: (1) impose safe state before resume; (2) resume before impose safe state. They reported the service disruption of the two algorithms on an Instant Messaging Service for adding or removing a fragmentation service. Their quantitative results showed that the latter had less disruption than the former because it exploited concurrency through the coexistence of the new and old sub-system.

2.3. *Impact analysis*

We next consider the most recent work on impact analysis on the QoS of RSURs. Feng and Maletic (2006) aimed at addressing dynamic change impact analysis at software architectural level. Given changes, they exploited dependence analysis to determine components, interfaces, and methods that would cause the impacts or were impacted by the changes. Actually, their analysis can be used to automate dynamic change planning, i.e., generating an operational reconfiguration plan from declarative change requirements. However, impact on the QoS of a RSUR was not discussed under the title of impact analysis of their article.

Morrison et al. (2007) proposed the Principle of Minimal Disruption (PMD) to state that the evolution of a software system should have minimal disruption to its execution. They proposed to apply PMD to the components that are selected as the scope of evolution, with the rest of the system continuing to execute concurrently. In fact in this case, a minimum disruption-based strategy is equivalent to localized evolution. PMD is a general principle to determine the minimal disruption by calculating when it is safe to perform evolution, what evolutionary steps are required, and how to execute the evolution operations. PMD generalizes the above as three factors: timing, nature, and means, which are treated as important in deciding the suitability of an evolutionary approach.

Fung and Low (2009) proposed an evaluation framework for dynamic reconfiguration. The scope of reconfiguration impacts and performance characteristics were included in the initial set of feature requirements. To refine these requirements, Fung and Low suggested that multiple version coexistence, performance characteristic prediction, dynamic state transfer, dynamic change impact analysis, and servicing continuity be the *principal criteria*. In their survey, the top ranked requirement was dynamic change impact analysis, which attracted considerable attention in composition-based distributed applications. In particular, the expected impact of dynamic changes was suggested by surveyed experts to be an important feature. A Wilcoxon signed-rank test was conducted to rank the importance of feature requirements, and dynamic change impact analysis ranked highest. An analysis of Fung and Low's results showed that the *principal criteria* were poorly supported by existing reconfiguration approaches. The strength of Fung and Low's framework lies in a qualitative confirmation of the desirable features for the evaluation of dynamic reconfiguration approaches.

In this article, we conduct an evaluation of available dynamic approaches against an evaluation benchmark that is sufficiently rich to exercise common problems that potentially affect the QoS of RSURs. Based on a comparative analysis, we present a clear evaluation of the impacts of dynamic reconfiguration. With such knowledge, an informed decision can be made as to the most appropriate approach to managing a particular reconfiguration scenario.

3. Evaluation criteria for QoS assurance capabilities

In this section, we propose a set of fundamental QoS characteristics that are used as the criteria to evaluate QoS assurance capabilities of dynamic approaches. The proposal is based on following requirements.

1. A QoS characteristic should hold during the normal execution of a system without disruption from dynamic reconfiguration.
2. A QoS characteristic should be sufficiently generic to cater for a variety of applications.
3. A QoS characteristic should be addressed by some reconfiguration approaches. Otherwise, it would be declined for a RSUR.
4. A QoS characteristic should be compatible with specific technologies to enable judgments about suitability of specific technologies for specific scenarios.

3.1. *Global consistency*

We discuss global consistency under the assumption that both the original system and the resulting system arising from reconfiguration are correct in their own right, apart from issues of reconfiguration. To verify a reconfiguration itself sane, we focus on the *transformation phase* while the reconfiguration is taking place.

Global consistency is concerned with globally distributed transactions, where each transaction is carried out by multiple distributed components that are all liable to be reconfigured. To clarify the key points, a running transaction is *correct* if the results that it produces are not modified by a reconfiguration. For example, a valid/invalid data-item from the original or resulting system needs also to be verified valid/invalid by the RSUR. A transaction is *complete* if it is not aborted during reconfiguration. That is, to be complete, once a transaction is started, it must be committed. The *critical state* of a running system is a set of states (distributed into a set of components) that are essential to the system's functional correctness. For example, the state of an encryption component prevents it from reproducing or reusing the session keys already used. A reconfiguration preserves a system's critical state if the state is transferred into the new sub-system before it is functioning. In the current example, if the encryption component is replaced, its state must be transferred into the replacing component. Otherwise, the reconfiguration reduces the security strength of the system because a previous session key could be reproduced and reused.

In addition, if we assume that the original system is defined by one application protocol and the resulting system is defined by another, transactions are then protocol-oriented, i.e. a transaction services for either the new or the old application protocol. A set of components is *dependent* if they service for a single application protocol and a transaction needs to be performed by all components of the set. A reconfiguration of a set of dependent components is termed a *dependent update*, described by Almeida et al. (2004) as follows: "an update of a component needs an update of another component, which relies on the former's behavior or other characteristics." A typical dependent update involves an encryption component and a decryption component. If the encryption component is updated with a new stronger encoder, the decryption component must be updated with a newer corresponding decoder. A transaction needs to be serviced by either the old encoder/decoder or the new encoder/decoder. Some typical dependent updates can be found in Mitchell et al. (1999) and Vandewoude et al. (2007). Based on the above discussion, global consistency is defined as follows.

Definition 1. A reconfiguration is *globally consistent* if it preserves system-wide transactional correctness and completion, critical state, and component dependency for a RSUR.

3.2. *Service availability*

We recognize that while a dynamic approach supports global consistency for the reconfiguration of a running system, it actually supports another characteristic: availability at the same time.

Definition 2. The service of a running system is still *available* if the system remains online and accepting clients' requests while reconfiguring.

Ensuring global consistency and also keeping a system available requires a means for constraining architectural and functional updates only to occur at a well understood safe state. A number of studies addressed the issue of how to lead a system into a safe state for dynamic reconfiguration. These studies include Ajmani et al. (2006), Almeida et al. (2004), Arshad et al. (2003), Bidan et al. (1998), Dowling and Cahill (2001a,b), Kim and Bohner (2008), Kramer and Magee (1990), Moser et al. (2002), Palma et al. (2001), Rasche and Polze (2008), Vandewoude et al. (2007), Warren (2000), Warren et al. (2006) and Zhang et al. (2004). Although these approaches vary in implementation, they apply the same concept of *quiescence*, proposed by Kramer and Magee (1990) and commonly treated as the most significant principle for preserving global consistency and availability. Dowling and Cahill (2001b) presented a typical protocol for quiescence, which uses the *freeze* operation to lead a system into a quiescent state before reconfiguration can occur. The freeze operation (sometimes called *blocking* or *suspending*) stops the part of a system necessary for updating and leads already started transactions to completion. The problem with quiescence is that although the system remains online to accept user requests, the affected parts of it do not process requests or respond to users while they are quiescent. Therefore, the workflow of a RSUR is actually stopped somewhere by quiescence.

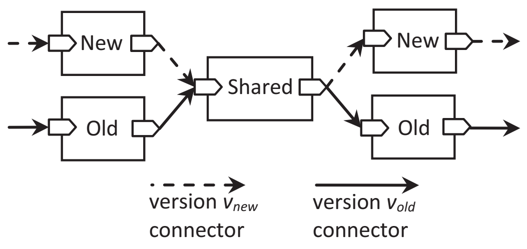

Fig. 1. DVM ensures independent workflow in a partially shared structure.

3.3. *3.3. Coexistence and service continuity*

We define coexistence as a stronger QoS characteristic than previous ones in the spectrum of QoS characteristics.

Definition 3. The new sub-system coexists with the old sub-system if the new sub-system is fully brought into effect by a reconfiguration before shutting down and then removing the old sub-system.

The coexistence characteristic is stronger than previously defined QoS characteristics because it is an essential condition for maintaining a running system's continuity in processing and responding to user requests (Mitchell et al., 1999). Based on coexistence, we define the continuity characteristic.

Definition 4. The service of a running system is continuing if the workflow of the system is never logically blocked by a reconfiguration such that the system is able to keep accepting and processing requests.

It is very rare that all components of a system are replaced by a reconfiguration. Therefore, the old and new sub-systems are not structurally independent but share some components during coexistence. Consequently, the coexistence characteristic brings with it a challenge: *a partially shared structure*. To provide service continuity, quiescence is not applicable and a novel measure is necessary. The key issue is to ensure *independent workflow in a partially shared structure*. The solution to this problem is *dynamic version management* (DVM). DVM is detailed in a previous publication (Li, 2009) and summarized as follows for component-based systems as illustrated in Fig. 1.

DVM uses the following elements as version carriers. The *system* as a whole is versioned as the global clock; both *connectors* and *application threads* (representing transactions) are versioned. A newly created thread always takes the current system's version and a thread's version remains unchanged throughout its lifecycle. Two versions (v_{old} and v_{new}) are sufficient for DVM because there are only two application protocols coexist during reconfiguration. The timing control of DVM is illustrated in Table 1. In normal (T_0), a system is running in v_{old} (single version of three carriers). In this status, a reconfiguration can start at any time by installing the new components and new connectors, of which the new connectors are versioned v_{new} directly. On completion of the installation, the system is brought onto status T_1, at which the system is switched from v_{old} into v_{new} for transformation. In this status, a newly created thread takes v_{new} and therefore the old and new application protocols coexist but are distinguished by two invocation chains of v_{old} and v_{new} respectively. At a shared component, an application thread makes a dynamic selection of an invocation chain by matching its own version with a connector's version. The existing v_{old} threads progress naturally and finish at T_2, where the old components and connectors can be removed except the old but reserved (used in the new sub-system) connectors are re-versioned v_{new}. After T_2 the system is running in v_{new} (single version of three carriers).

Table 1
DVM timing control.

Version carrier		T_0 the system is in normal running before a reconfiguration	T_1 the new sub-system is ready	T_2 the v_{old} transactions commit
System (v_s)		v_{old}	$v_s = v_{new}$	v_{new}
Connectors (v_c)	Existing and reserved	v_{old}	v_{old}	v_{new}
	Existing but to be removed	v_{old}	v_{old}	
	Newly installed		$v_c = v_{new}$	v_{new}
Threads (v_t)	Created before T_1	$v_t = v_s$	v_{old}	
	Created after T_1		$v_t = v_s$	v_{new}

3.4. *State transfer*

State transfer is an invisible, obscure factor which nevertheless could impair QoS of a RSUR considerably.

Stateful components work correctly if their states are persistent. For example, in Janssens et al. (2007), to replace a component that generates unique IDs, the replacing component should not repeat IDs that were already generated by the original component. To preserve this feature, the replacing component must be initialized into a suitable state. Generally, if a stateful component is replaced, its state must be transferred into the replacing component before the latter is allowed functioning. State transfer is to fulfill the requirement of preserving system-wide critical state so that components' states can survive reconfiguration.

State transfer is a requirement for QoS assurance of dynamic reconfiguration. First, state transfer must be performed under well understood circumstances. Otherwise, transfer could modify the results of running transactions or cause them to fail. A direct and effective mechanism is to apply quiescence: it is known safe to make a state transfer when the affected part of a system is quiescent (Vandewoude et al., 2007). However, the drawback is that the system performance experiences a serious decline when it is quiescent because the new sub-system cannot function till state transfer is complete and it can then be re-activated.

Second, the coexistence characteristic can just ensure service continuity in logic but could bring another problem: *unintentional blocking*. In a concurrent environment, a thread servicing the new sub-system can run faster than a thread servicing the old sub-system. However, the former is unintentionally blocked while it invokes a replacing component, which is waiting for a state to be transferred from the replaced component that is still acting. To illustrate the unintentional blocking problem, we base on DVM for a discussion. 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 v_{old} are committed, where $t_s \leq t_c$. Existing stateful systems (Ajmani et al., 2006; Dowling and Cahill, 2001a; Hicks et al., 2005) do state transfer at t_c. The rationale is that when components are no longer used by any transactions of v_{old}, state transfer is safe. 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 concurrent environment, a transaction of v_{new} could progress faster than a transaction of v_{old}. Consequently, the replacing component is likely already waiting for the state transfer between t_s and t_c, However, the state transfer is delayed until t_c to preserve consistency. State transfer at t_c is safe but too conservative and could cause *unintentional blocking* of workflow.

For example, in Fig. 2, C is a shared component, and C_1 and C_3 are unshared components. C_1 is to be replaced by C_2, and the state of C_1 needs to be transferred into C_2. We cannot predict when to serialize C_1's state and deserialize the state into C_2 such that unintentional blocking does not occur. This is because architectural and functional concerns should be separated, and therefore the inner functionality of C and the data being processed should not be inspected by a reconfiguration framework.

As a physical factor, state size affects QoS. The size of state varies from component to component. If state size is large, state transfer by a deep copy of state variables sometimes needs considerable CPU time and causes threads to wait, i.e. unintentional blocking.

Consequently, by coexistence, a system's service is kept logically continuing but may be physically blocked by physical factors such as state transfer. The QoS of a RSUR is then affected as a result of blocked workflow. Therefore, stateful components are required to expose the QoS problem of state transfer. To differentiate dynamic approaches on their capabilities for state transfer, we define stateless equivalence as a QoS characteristic.

Definition 5. A stateful system is made stateless-equivalent by a reconfiguration if state transfer does not cause any suspension of the application workflow.

3.5. *Physical overhead*

In this section, we analyze other physical factors that could expose impacts on the QoS of a RSUR, and on this basis define the strongest QoS characteristic in the spectrum of QoS problems.

3.5.1. *Reconfiguration overhead*

Reconfiguration overhead is a factor, which could physically degrade QoS considerably. While a reconfiguration executes concurrently with ongoing transactions on the same physical machine, it can suspend ongoing transactions due to temporary reallocation of resources (CPUs, memory, or network bandwidth). The issue of runtime overhead of a reconfiguration has been recognized by Ajmani et al. (2006), Evans (2004), Hicks et al. (2005), Mitchell et al. (1999), Rasche and Polze (2008), and Truyen et al. (2008). However, these do little to address the problem. Where overhead is not considered, a reconfiguration may fully compete for resources. The competition capacity of a reconfiguration will have different impact on the QoS of a RSUR in different situations, depending on resource availability, but can be controlled in all situations. However, full competition capacity is not always necessary, and there is a way to minimize the impact of reconfiguration overhead. These features are further discussed as follows.

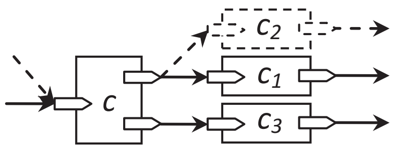

Fig. 2. An example structure in which it is impossible to predict the right time to make a state transfer from C_1 into C_2 such that unintentional blocking does not occur.

3.5.2. *Resource availability*

Resource availability is a physical factor that affects the use of a reconfiguration approach and therefore the QoS of a RSUR. While the normal workload is differentiated by the ongoing transactions from system to system or from time to time for a particular system, the normal usage of the system's resources is thereby different. For example, if the computing load of ongoing transitions is high, the CPU usage will be high and possibly saturated (on 100% usage). If the CPU is always saturated by ongoing transactions, reconfiguration must affect QoS. The reason is that reconfiguration must compete for a certain amount of resource with ongoing transactions. Consequently, a key issue of designing a reconfiguration algorithm is the controllability of resource usage. However, if the CPU is idle (below 100% usage) from time to time, a reconfiguration algorithm can be designed to make use of only free CPU time and avoid affecting the QoS of a RSUR. In general, a reconfiguration approach may present different impacts under different resource availabilities. From another perspective, a reconfiguration approach may be specially designed to work with a designated resource availability so that QoS can be taken into account.

To take the above physical factors into account, we define the strongest QoS characteristic as follows.

Definition 6. A running system is QoS assured if the system's QoS is physically maintained not lower than a pre-determined baseline during a reconfiguration.

4. Classification of reconfiguration approaches

While reconfiguration approaches can be seen from different perspectives, we classify them by the QoS characteristics defined in the previous section. On reviewing the current literature, first, each article has been checked for what technology is used to preserve application consistency, because consistency is the fundamental requirement to be fulfilled by all dynamic approaches. If a blocking operation is used by an approach, it is classified as consistency and availability for QoS assurance. This is because the approach cannot support stronger QoS characteristics (Sections 3.3–3.5) once a blocking operation is used. Second, if a system's service is never logically blocked, the approach must support the coexistence of the old and new sub-system. The key technology used by the approach must be a sort of dynamic version management (Section 3.3). The classification is summarized in Table 2.

From Table 2, we can see that most approaches ensure availability by applying quiescence, or ensure continuity by applying some forms of dynamic version management. However, we found that existing reconfiguration approaches, except for our model (Li, 2009) or this article, do not support QoS characteristics that are stronger than service continuity. Some supporting evidences are: as a typical approach to consistency and availability, Kim and Bohner (2008) investigated the use of aspect-oriented programming to separate reconfiguration concerns from functional concerns to facilitate dynamic reconfiguration. Kim and Bohner modeled dependent updates, cyclic method calls and state transfer using aspect-oriented facilities. Their approach applies quiescence to ensure application consistency, and therefore a blocking operation has to be used to drive an aspect to a safe status for a reconfiguration. To deal with dependent updates, the approach requires that a replacing component must provide the same interfaces as those of a replaced component. A component's state is transferred by a deep copy of the state variables from the replaced component into the replacing component.

With respect to continuity, Mitchell et al. (1999) proposed a coexistence model, which allows new components to start running before the removal of old components. However, because there is no dynamic version management, the model relies on *precise timing* (the processing latency of each component and the start-up time of each new component) to determine when to stop the replaced components and when to start the replacing components. Under such precise scheduling, the switching of a multimedia stream from the initial processing chain to the resulting processing chain fulfils QoS constraints. In short, to some extent, the model exploits the coexistence characteristic in support of QoS assurance.

The classification deserves two further notes. First, the classification is at a level that is able to distinguish/classify reconfiguration approaches to the granularity that groups together related work with the same QoS assurance capabilities. For example, we do not further differentiate *tranquility* (Vandewoude et al., 2007) from *quiescence* (Kramer and Magee, 1990) because we assume that quiescence and tranquility have similar quality and the difference between them is trivial. They both must block the application workflow in attaining quiescence or tranquility; quiescence is a fallback after tranquility because the latter is not guaranteed to be reachable in bounded time. Second, some approaches such as Hillman and Warren (2004a,b), Janssens (2006), Janssens et al. (2007), Li (2009) and Truyen et al. (2008) are classified into multiple groups because to some extent these approaches are open and support different algorithms with different levels of QoS capabilities.

The remainder of this article conducts an evaluation to quantitatively compare the QoS assurance capabilities of existing dynamic reconfiguration approaches and to confirm the above qualitative classification.

5. Discovery criteria for QoS assurance capabilities

In this section, we propose criteria for discovering the QoS assurance capabilities of a reconfiguration approach. These criteria form the basis for a quantitative evaluation of dynamic reconfiguration approaches.

5.1. *Global consistency discovery*

The verification of global consistency is application dependent. To minimize this dependence in a generic evaluation, the following criteria are proposed to exploit simple conditions easily fulfilled by domain applications.

Definition 7. A reconfiguration is identified as *consistency-preserved* if no incorrect results (application-specific) or mismatches (between the number of submitted requests and the number of results that are created) are identified during the reconfiguration period.

To explain this criterion, an incorrect result implies workflow interleaving between the new and old sub-system, or failure of state transfer. The number mismatch between requests and results implies transaction abortion.

Although possession of the above feature remains application-dependent, designing an evaluation benchmark with the above feature does not force the application of any specific constraints to a particular reconfiguration approach.

5.2. *Quiescence discovery*

To ensure global consistency and provide service availability, most existing approaches must apply quiescence. However, as pointed out by Vandewoude et al. (2007), quiescence disturbs QoS significantly.

Definition 8. A reconfiguration is identified as *quiescence-applied* if the QoS of a RSUR experiences a sharp fluctuation (decline significantly and re-bounce to normal).

Table 2
The classification of reconfiguration approaches by the QoS characteristics.

QoS characteristic	Reconfiguration approach
Consistency and availability (quiescence or tranquility is applied)	List-1: Ajmani et al. (2006), Almeida et al. (2004), Arshad et al. (2003), Bidan et al. (1998), Dowling and Cahill (2001a,b), Hillman and Warren (2004a,b), Janssens (2006), Janssens et al. (2007), Kim and Bohner (2008), Kramer and Magee (1990), Leger et al. (2010), Moser et al. (2002), Palma et al. (2001), Rasche and Polze (2008), Surajbali et al. (2010), Truyen et al. (2008), Vandewoude et al. (2007), Warren et al. (2006) and Zhang et al. (2004)
Continuity (logically ensured by the coexistence of the new and old sub-system)	List-2: Ajmani et al., 2006; Cook and Dage, 1999; Evans, 2004; Hicks et al., 2005; Hillman and Warren, 2004a, 2004b; Janssens, 2006; Janssens et al., 2007; Mitchell et al., 1999; Senivongse, 1999; Solarski and Meling, 2002; Tewksbury et al., 2001; Truyen et al., 2008
Continuity plus stateless equivalence	Li (2009); this article
QoS-assurance (applicable to CPU non-saturation)	Li (2009); this article
QoS-assurance (applicable to CPU saturation)	Li (2009); this article

A qualitative investigation of the existing approaches shows that quiescence is the one to cause the sharpest QoS fluctuation among all consistency algorithms. Although quiescence is just a sufficient condition and there are probably other conditions that could cause a sharp QoS fluctuation, a review of the current literature reveals no other such conditions in the existing approaches. Consequently, a sharp fluctuation can be safely treated as a sufficient condition currently to discover quiescence for evaluating the existing approaches.

5.3. Service continuity discovery

Under this condition of logical continuity of service, quiescence is not applicable to the preservation of global consistency because blocking workflow logically conflicts with service continuity. DVM is thus the minimum requirement for preserving both consistency and continuity. To apply DVM, transactions are versioned for the coexistence period. Independent workflow requires that the old sub-system services transactions from one version while the new sub-system services transactions from the other version. DVM then ensures independence between two versions of transactions while they are concurrently running in a partially shared structure.

Definition 9. A reconfiguration is identified as *continuity-maintainable* if the QoS of a RSUR changes smoothly without a sharp or abrupt fluctuation.

Related to the QoS feature of quiescence, the removal of sharp fluctuation currently signifies both the removal of quiescence and the use of coexistence in support of service continuity.

5.4. Stateless equivalence discovery

State transfer may cause unintentional blocking in a concurrent environment, namely when transactions are blocked at a replacing component that is waiting for a state to be transferred from a replaced component, which is in processing and not safe for state transfer. The solution to unintentional blocking is to convert a stateful system into a virtually stateless equivalent. We argued in Section 3.4 that it is impossible to predict precise timing for making such a conversion unless it is application-dependent and at the cost of mixing application concerns with reconfiguration concerns. For example, Mitchell et al.'s (1999) approach may be extended to avoid the unintentional blocking problem of state transfer. However, we expect that relying on precise timing to support the coexistence of the old and new sub-system makes the extension application-dependent and requires runtime inspection of component implementations and data being processed. Consequently, the application and reconfiguration concerns cannot be separated.

A generic solution to stateless equivalence is state-sharing: allowing the replaced component's state to be shared by the replacing component. State-sharing requires the following constraints to be applied.

1. Compatibility of state format between the replaced component and the replacing component.
2. Encapsulation of a component's state variables into a single wrapper object.
3. Mutual exclusion of access to a shared state.
4. Permission of such sharing by application logic.
5. Replacing and replaced components reside on the same physical machine.

The first two constraints are easily fulfilled by any domain applications. Constraint 3 ensures the atomicity of the state update and no corrupted state. Constraint 4 implies that state-sharing is still application-dependent, and therefore it is not applicable to every application. However, we assume it can cater for a variety of applications. For instance, identifier generators or number counters are simple examples to permit such state-sharing. For constraint 5, although component state migration is an open issue in this article, 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.

Once the above constraints are fulfilled, state-sharing is applicable, and we obtain the following benefits: (1) the state is always available when a replacing component is invoked; (2) regardless of whether the state is being used by the replaced or the replacing component, the processing is contributing to the overall QoS of a RSUR.

Definition 10. A reconfiguration is identified as *stateless-equivalent* if a change of the number of reconfigured stateful components, or a change of the state size of a stateful component, has no impact on the QoS of a RSUR.

We assume that state-sharing can enable state transfer virtually instantaneously and achieve stateless equivalence because state-sharing policy can be applied before the coexistence period and a state-sharing can be done by redirecting a single reference (of the wrapper object).

5.5. Discovery of overhead controllability

We assume that a reconfiguration possesses all the previously defined characteristics (global consistency, availability, continuity, and stateless equivalence). Under such an assumption, we can filter out other factors that could affect QoS and focus on the impact of reconfiguration overhead. Also under such an assumption, if overhead is controlled, QoS is assured. Consequently, QoS assurance is a synonym of overhead control under this assumption and treated as the strongest characteristic in the spectrum of QoS problems.

As established in Section 3.5, reconfiguration overhead can impair the QoS of a RSUR significantly, and control of overhead must be addressed in consideration of resource availability that is mainly reflected onto CPU usage. Consequently, while discussing the controllability of overhead, the following criteria are relevant.

Definition 11. For CPU non-saturation, a reconfiguration is identified as *QoS-assured* if it has no impact on the QoS of a RSUR.

For CPU saturation, reconfiguration overhead must have impact on the QoS of a RSUR. However, whether a reconfiguration has controllability of overhead depends on whether it has controllability of its own CPU usage.

Definition 12. For CPU saturation, a reconfiguration is identified as *QoS-assured* if it can be restrained for a predetermined CPU usage and the QoS of a RSUR can be kept not lower than a predetermined baseline.

6. Evaluation benchmark

We now introduce an evaluation benchmark to expose existing QoS problems. If a reconfiguration approach is inherently incapable of coping with a QoS problem, we can identify it by applying the evaluation benchmark and the discovery criteria (Section 5) against the approach. Therefore, the design of an evaluation benchmark should fulfill the following requirements.

Definition 13. An evaluation benchmark is *consistency testable* if it accommodates a partially shared and partially independent structure for a reconfiguration.

Definition 14. An evaluation benchmark is *critical-state testable* if it accommodates stateful components which are to be replaced by a reconfiguration.

Definition 15. An evaluation benchmark is *overhead testable* if the computing and communication loads of a reconfiguration are application-independent and settable.

Definition 16. An evaluation benchmark is *workload* (*CPU usage*) *testable* if the application workload (computing, communication and clients' invocations) is application-independent and settable.

In addition, while reconfiguration approaches are evaluated, they should be compared quantitatively. Therefore, another requirement is needed.

Definition 17. An evaluation benchmark is *QoS measurable* if the application scenario is able to facilitate a quantitatively statistical characterization of the throughput and response time.

6.1. Conceptual model

The conceptual model for the evaluation benchmark is a *secured data transmission* system, named *DEDSS* (*Data Encryption and Digital Signature System*). The model ensures the security of data transmission by encryption and digital signature. DEDSS uses both symmetric and asymmetric encryption algorithms, and its application scenario is as follows.

1. The system consists of a single sender and multiple receivers. The sender has the *public keys* of all receivers, and each receiver has the *public key* of the sender.
2. Before sending a data-item, the sender generates a *session key*. The data-item is then encrypted with the session key using a *symmetric algorithm*, and digitally signed with the sender's *private key* using an *asymmetric algorithm*.
3. The session key is then encrypted with a receiver's public key using an *asymmetric algorithm*.
4. The encrypted session key and the encrypted and digitally signed data-item form a *data package* to be transmitted.
5. When the receiver receives a data package, it decrypts the session key with its own private key.
6. The receiver then uses the session key to decrypt the data-item and verifies the data-item's digital signature by the sender's public key.
7. The receiver is able to verify *confidentiality*, *authentication* and *integrity* of the transmitted data-item if both the decryption of the data-item and the verification of the digital signature are successful.

Based on the proposed criteria in this section, DEDSS is an effective evaluation benchmark:

1. *Stateful system*: The encryption or decryption keys comprise the states of the encryption, decryption, signing, and signature verification components at the sender or receivers. Since a session key is non-repeatable, the encryption component needs to maintain state which prevents it from repeating session keys already produced.
2. *Facilitating varying workload*: The computing load of components, the size of a state, and the computing load of instantiating or removing components, and the invocation load from clients are adjustable without altering the application scenario. This feature allows simulation of maintaining a large size of state.
3. *Dependent components*: The sender's encryption component and receiver's decryption components must follow the same protocol. The hash algorithm used by both ends must be the same. Any change results in dependent updates. Furthermore, the model allows introduction of new dependence. For instance, if compression is to be used, the compression and decompression components are interdependent.
4. *Independent* (*shared*) *components*: There are protocol-independent components for packing data-items and sending, receiving and unpacking data-packages. During the coexistence period, these independent components are shared by the new and old sub-systems. Thus, they create a partially shared structure between different application protocols.
5. *Concurrency*: Each service call will initiate a processing thread. All concurrent threads have the same execution priority.

A reconfiguration scenario of DEDSS proposed by this article for the evaluation consists of the following reconfiguration tasks:

1. Dependent updates of encryption and decryption components (state transfer; replacement of symmetric and asymmetric encryption and decryption algorithms), message digest components (replacement of hash algorithms), and signing and signature verification components (state transfer; replacement of asymmetric encryption and decryption algorithms).
2. Dependent updates to add new components for message compression (before encryption) and decompression (after decryption).

These are an actual series of study cases to demonstrate the capabilities of their systems (Ajmani et al., 2006; Bidan et al., 1998; Hillman and Warren, 2004a,b; Truyen et al., 2008; Vandewoude et al., 2007), but DEDSS is more complex, being a stateful system, having a partially shared structure, rich application and reconfiguration scenarios and varying workload, albeit artificial.

To verify the effectiveness of conceptual DEDSS, two architectural prototypes have been implemented on DynaQoS (Dynamic Reconfiguration for QoS Assurance), a dynamically reconfiguring framework that we have developed in Java (Li, 2009), and thereby

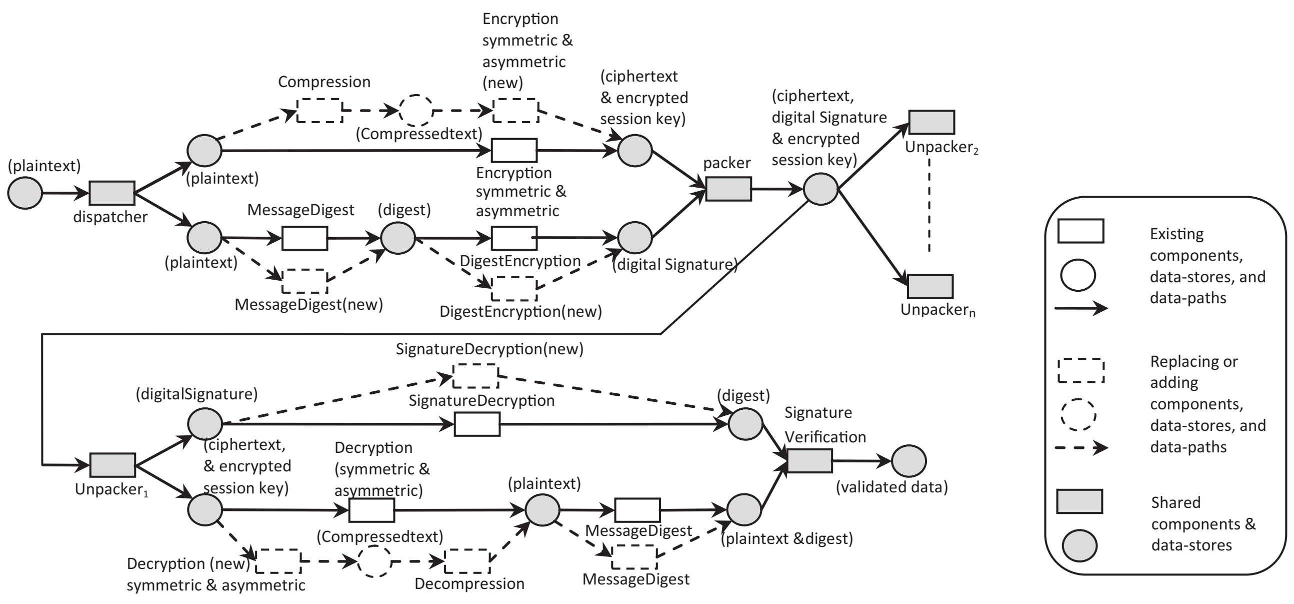
Encryption symmetric & asymmetric (new)
Compression
(ciphertext & encrypted session key)
(ciphertext, digital Signature & encrypted session key)
(Compressedtext)
(plaintext)
dispatcher
(plaintext)
Encryption symmetric & asymmetric
packer
Unpacker2
MessageDigest
(digest)
(plaintext)
DigestEncryption
(digital Signature)
MessageDigest(new)
DigestEncryption(new)
Unpackern
SignatureDecryption(new)
(digitalSignature)
SignatureDecryption
(digest)
Signature Verification
(ciphertext, & encrypted session key)
Decryption (symmetric & asymmetric)
(plaintext)
Unpacker1
(validated data)
(Compressedtext)
MessageDigest
(plaintext &digest)
Decryption (new) symmetric & asymmetric
Decompression
MessageDigest
Existing components, data-stores, and data-paths
Replacing or adding components, data-stores, and data-paths
Shared components & data-stores

Fig. 3. F-DEDSS, the dataflow prototype of conceptual DEDSS.

exploring the fundamental software architectures for dynamic reconfiguration and impact analysis on the QoS of RSURs.

6.2. Dataflow prototype

The dataflow prototype (F-DEDSS) of conceptual DEDSS is illustrated in Fig. 3. F-DEDSS is an extension of Dataflow Process Network (Lee and Parks, 1995; Najjar et al., 1999) with dynamically reconfigurable capability. F-DEDSS consists of functional components (rectangles), data-stores (circles) and data-paths (directed lines). A *functional component* accepts data-items through its *entrances*, processes them, and delivers results through its *exits*. A *data-store* is conceptually a random-accessible data buffer with infinite capacity. A *data-path* is a connector between a functional component and a data-store, through which the component can either access the data-store to consume data-items or deliver the produced results to the data-store. For a functional component, the consumption of data-items is controlled by a *firing rule*. The firing rule indicates a condition that data-items are ready for processing, and a functional component is automatically triggered when its firing rule is satisfied.

The reconfiguration scenario is also illustrated in Fig. 3. The dashed part (rectangles, circles and directed lines) consists of the replacing or newly added functional components, data-stores and data-paths. The components that are bypassed by the new components are to be removed. The scenario shows that the new and old sub-systems form a partially shared structure. Independent workflow must be ensured for the coexistence period. Consequently, the key issues to implement DVM for F-DEDSS are:

1. independent components must consume only single version of data-items throughout the coexistence period;
2. shared components can consume two versions of data-items but must consume only a single version of data-items in a single firing.

It should be mentioned that although the reconfiguration scenario in Fig. 3 covers components' replacement and addition, DVM of DynaQoS supports component removal as well because DVM treats a replacement as a removal plus an addition. A technical summary of DVM is presented in Section 3.3 and the details of DVM can be found in a previous publication (Li, 2009).

6.3. Component prototype

The component-based prototype of conceptual DEDSS is illustrated in Fig. 4.

C-DEDSS is an extension of a traditional component model with dynamically reconfigurable capability. In a component-based model, each architectural element encapsulates its functionalities as a black box and publishes a set of services offered to other architectural elements (Soria et al., 2009). As illustrated in Fig. 4. C-DEDSS consists of components (rectangles) and connectors (directed lines). Each *component* has *provided-service* and *required-service* interfaces, which provide services for or require services from other components. A *connector* links a required-service of an invoker component to a provided-service of an invoked component, i.e. component dependencies are expressed locally and abstractly, and composition makes them explicit. As a result, coordination among services is explicitly interpreted by connectors. Connectors make components loosely coupled, and therefore an architectural reorganization can be achieved through changes of connectors at runtime. The proposed reconfiguration scenario is also illustrated in Fig. 4. The dashed part (rectangles and directed lines) consists of the replacing or newly added components and connectors. The components that are bypassed by the new components are to be removed. The scenario shows that the new and old sub-systems form a partially shared structure. Independent workflow must be ensured for the coexistence period. Consequently, the key issues to implement DVM for C-DEDSS are:

1. connectors are versioned to distinguish the two networks of components, of which some are shared by the old and new subsystem for the duration of coexistence;
2. an application thread is versioned to service for only a single protocol: either the old or new sub-system;
3. at a shared component, an application thread chooses the connector whose version matches its own for forwarding dependent invocations.

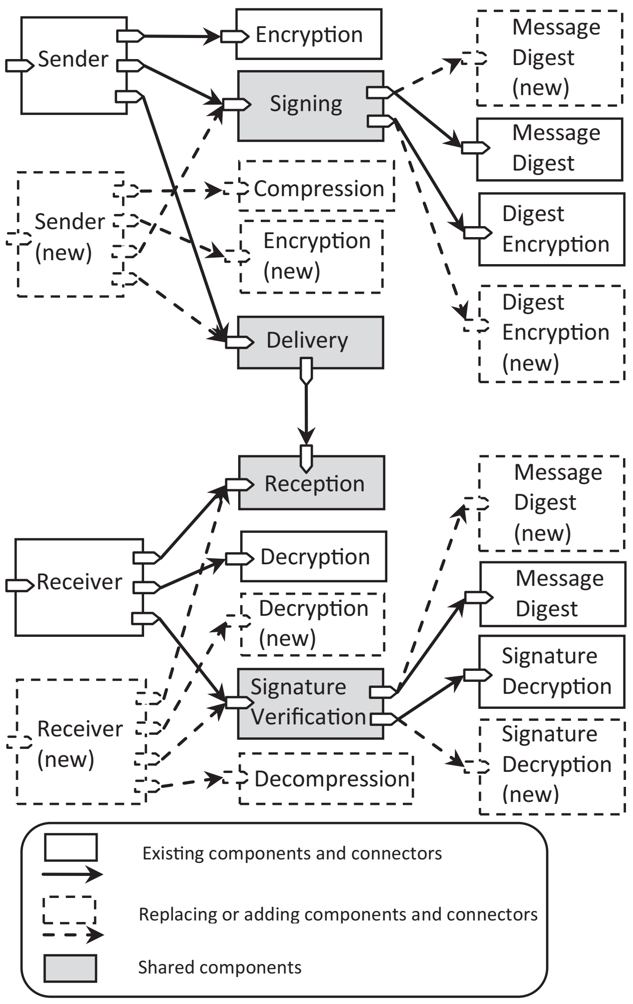

Fig. 4. C-DEDSS, the component-based prototype of conceptual DEDSS.

7. The evaluation context

To evaluate reconfiguration approaches and compare their QoS assurance capabilities, a model is needed to abstract and realize these approaches so that they can be tested on a unified context for a sound comparison. In this section, we describe such a design of a unified evaluation context.

7.1. Realization of reconfiguration approaches

To realize (represent) reconfiguration approaches, the following two types of abstraction are necessary and effective in our practice.

1. *Representation of logical control for QoS*: The abstraction should permit four options: use of quiescence; use of DVM; state transfer by deep copy of state-variables; state transfer by state-sharing.
2. *Representation of physical control for QoS*: The abstraction should permit three options: full competition; no competition, and controlled competition.

The above options are effective for representing existing approaches, and the granularity of the options is able to group together related approaches with the same QoS assurance capabilities. On DynaQoS, the options are realized by reconfiguration *planners* and *schedulers*, where planners realize logical control and schedulers realize physical control respectively. We have implemented three planners.

1. The *availability planner* realizes quiescence and state-variable copy options. This planner applies quiescence to preserve application consistency, and therefore the blocking operation is applied. Component state is transferred by a deep copy of state variables when the system is quiescent. This planner ensures both application consistency and service availability of a RSUR.
2. The *continuity planner* realizes DVM and state-variable copy options. DVM ensures safe coexistence of the old and new sub-system and logical continuity of service of a RSUR, but component state is still transferred by a deep copy of state variables.
3. The *stateless-equivalence planner* realizes DVM and state-sharing options. This planner still applies DVM, but state transfer is through state-sharing by redirecting the state's reference of the replaced component into the replacing component.

We have implemented three schedulers.

1. The *competition scheduler* realizes the full competition option. This scheduler assigns a reconfiguration thread the same execution priority as that of ongoing transactions, and therefore gives full competition capability to the reconfiguration thread.
2. The *pre-emptive scheduler* realizes the no competition option. This scheduler enables ongoing transactions always to pre-empt a reconfiguration thread, and therefore disables competition capability of the reconfiguration thread. Consequently, this scheduler is applicable to CPU non-saturation only.
3. The *time-slice scheduler* realizes the controlled competition option. This scheduler is design for CPU saturation case, i.e. CPUs are always saturated by ongoing transactions. This scheduler assigns the same execution priority to both a reconfiguration thread and ongoing transactions, and therefore allows the reconfiguration thread to be able to execute concurrently with the ongoing transactions. To realize the controlled competition option, this scheduler divides the scheduling time into schedulable time-slices, for example 1000 ms time-slices. In each time-slice, the ongoing transactions are always runnable, but the reconfiguration thread is made runnable only for a pre-determined time-slot, for example 200 ms time-slots. The runnable time-slots enable the reconfiguration thread to have only 20% chance competing for CPUs, and therefore this scheduler executes the reconfiguration thread with controllable overhead.

When the two types of tuner (planner and scheduler) are ready, the realization of a reconfiguration approach is to combine reconfiguration planners with schedulers. Each *effective* combination is termed a *reconfiguration strategy*, which is the representation of a reconfiguration approach onto DynaQoS with its inherent QoS characteristics embedded (Table 3).

The realization of available reconfiguration approaches respects their original QoS characteristics. First, QoS assurance has only recently received significant attention, and therefore the majority of related approaches are not QoS-oriented. Furthermore, we realize related approaches according to the granularity defined by the QoS characteristics (Section 3). Consequently, they are realized by two basic strategies: *avl-cpt* and *con-cpt*. Third, although related approaches in the same group vary from one to another, the variations are trivial in terms of the defined QoS characteristics, and do not need further differentiation in this evaluation. Otherwise, the differentiation will not be in line with the discussion mainstream, and the refinement will make little difference in terms of the QoS characteristics. For example, we realize tranquility and quiescence by the same strategy: *avl-cpt*, because tranquility has no essential

Table 3
The reconfiguration strategies as a realization of available reconfiguration approaches.

Strategy	Planner and scheduler	QoS characteristics ensured	QoS assurance techniques applied	Reconfiguration approach realized
avl-cpt	Availability planner and competition scheduler	Global consistency and service availability	Quiescence or tranquility	List-1 in Table 2
con-cpt	Continuity planner and competition scheduler	Global consistency, service availability and service continuity	DVM and full competition	List-2 in Table 2
sle-cpt	Stateless-equivalence planner and competition scheduler	Global consistency, service availability, service continuity and stateless equivalence	DVM, state-sharing and full competition	Li (2009) and this article
qos-pre	Stateless-equivalence planner and pre-emptive scheduler	Global consistency, service availability, service continuity, stateless equivalence and overhead control (applicable to CPU non-saturation)	DVM, state-sharing and no competition	Li (2009) and this article
qos-ts	Stateless-equivalence planner and time-slice scheduler	Global consistency, service availability, service continuity, stateless equivalence and overhead control (applicable to CPU saturation)	DVM, state-sharing and controlled competition	Li (2009) and this article

difference from quiescence because (1) tranquility is not always reachable and must use quiescence as a backup; (2) both cause a sharp QoS decline due to the use of blocking operations. Their difference is in quantity because tranquility minimizes the affected part of the system, but not in quality because both must block the workflow.

7.2. Implementation issues

7.2.1. Flexibility

DynaQoS is implemented as a flexible and open framework because it separates reconfiguration algorithms (termed strategies) from the reconfiguration framework and supports the *integration* of different reconfiguration strategies. 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). In that sense, DynaQoS is flexible for evaluating newer reconfiguration algorithms once these are available.

7.2.2. Timing control

To support coexistence, DVM is the key technique to ensure independent workflow on a partially shared structure between the old and new sub-system. We model DVM by applying timing control to the following three reconfiguration phases.

1. The *installation* phase loads and instantiates the new components (including state-sharing if applicable) and sets up the new connectors.
2. The *transformation* phase switches workflow from version v_{old} to v_{new}, and traces the ongoing transactions of version v_{old} to complete.
3. The *removal* phase finalizes and garbage-collects old components and deletes old connectors.

Timing control ensures that a reconfiguration can only occur at well understood time points and the certainty of its impacts on the QoS of a RSUR. Timing control uses the system, connectors and application threads as version carriers. The system's version is used as the global clock for timing control, and the connectors' versions coordinate with threads' versions to create independent workflow on a partially shared structure. DVM is briefly presented in Section 3.3, and the details of DVM can be found in a previous publication (Li, 2009) for further interest.

7.2.3. Implementation soundness

The proposed reconfiguration strategies (Section 7.1) are balanced except for the strategy: *con-cpt* that realizes List-2 (in Table 2) reconfiguration approaches and actually enlarges these approaches' capabilities for dynamic version management. For example, in support of the coexistence characteristic, Janssens (2006) installed the marking components (MCs) and dispatching components (DCs), which mark the packets from the old or new services and delegate the marked packets to the right services. However, such a packet-distinguishing mechanism incurs further issues besides the overhead of adding or removing MCs and DCs. Addition and removal of MCs and DCs introduces an extra distributed dependence that must be coordinated for a reconfiguration to succeed. In Janssens (2006), installation and removal of MCs and DCs requires blocking operations to impose safe state. That feature brings the model back to quiescence, which prevents it from having full coexistence. However, our DVM does not have such a problem supporting of the coexistence characteristic, i.e. although some approaches (List-2 in Table 2) have supported dynamic version management in different ways, when realizing existing approaches by DVM as described previously, we do not include any factors that could reduce the QoS strength for related approaches.

7.3. QoS measurements

As detailed in Section 2, existing evaluations are mainly based on the measurement of reconfiguration time (Bidan et al., 1998; Hillman and Warren, 2004a,b; Ajmani et al., 2006; Truyen et al., 2008; Leger et al., 2010; Surajbali et al., 2010). However, it is uncertain whether reconfiguration time is always important for evaluation. Suppose that a reconfiguration lasts a very short period but causes a very sharp decline on QoS, while another reconfiguration lasts for a relatively longer period but causes smaller impacts on QoS. If it is urgent to fix a bug, the former could be better, and maybe sometimes shutting down the system is necessary. However, if QoS is a major concern and a reconfiguration is to upgrade the system, the latter is more likely to be acceptable. On considering what impacts of a reconfiguration will finally reflect on the direct experiences of end-users, the common QoS values: *system throughput* and *response time* should be the key metrics. In our evaluation, we only 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 a request submission and its completion as confirmed by its response.

Throughput and response time comparison brings two benefits: (1) the QoS of a RSUR is soundly judged by these widely used metrics; (2) the evaluation results are reliable and reproducible if a positive correlation between throughput and response time is confirmed for a reconfiguration strategy.

8. Evaluation and result analysis

In this section, we evaluate the QoS assurance capabilities of available reconfiguration approaches on DynaQoS. We have developed DynaQoS using Java language and therefore it is able to run on any machine supporting a JVM (Java Virtual Machine). However, tests of thread scheduling policy show that Java threading policy is different on multiple-processor machines and on single-processor machines even if the same version of an operating system is used. To verify the effectiveness of reconfiguration strategies (as a realization of available reconfiguration approaches as described in Section 7.1) and the reliability of evaluation results, we conduct the evaluation on two machines. On both machines, the same version of Microsoft® Windows® XP operating system and the same Java SE 6 runtime provide a software platform.

1. Two-processor machine: Intel® Core™ 2 Duo, CPU 4300 @ 1.80 GHz and 1.79 GHz, and 2 GB RAM
2. Single-processor machine: Intel® Pentium® 4, CPU @ 3.00 GHz, and 1 GB RAM

Another measure towards reliable evaluation is to test the reconfiguration strategies on both F-DEDSS and C-DEDSS, which have different architectures. If the same results can be reproduced in both, the reliability of results is further confirmed. In this section, we report the result of running C-DEDSS on the two-processor machine, and F-DEDSS on the single-processor machine.

8.1. Evaluation of global consistency

The application scenario of DEDSS implies that global consistency is verified if:

1. the number of requests (data-packages to be sent) matches the number of results (data-packages to be received), and
2. each encrypted and signed data-package is successfully decrypted and signature-verified.

The above condition-1 can verify transactional completion in that no transactions have been aborted by a reconfiguration; condition-2 can verify that transactional correctness, component dependency, and system-wide critical state are preserved.

In about 50 tests, all the reconfiguration strategies have fulfilled the two conditions above. This confirms the effectiveness of reconfiguration strategies as a realization of dynamic approaches.

8.2. Evaluation of QoS

In this section we explore the effect of the various reconfiguration strategies on QoS for combinations of [saturation, throughput], [saturation, response time], [non-saturation, throughput], and [non-saturation, response time] for both F-DEDSS and C-DEDSS. Each combination is termed a *Test Context* (*TC*). Each reconfiguration strategy is tested separately on each TC, but all reconfiguration strategies are illustrated for the same TC in Figs. 5–12 respectively.

To present a clear comparison, without the involvement of reconfiguration, the running system's throughput and response time has been displayed for the original (denoted as *origin* in the charts) and the resulting (denoted as *target*) system as the comparison baselines.

The parameters in the conducted TCs were intentionally set differently. The variation was to test whether results were reproducible for different situations. For example, strategy *qos-ts[1000,500]* and *qos-ts[1000,200]* use the same 1000 ms time-slice but 500 ms and 200 ms time-slots respectively. The parameters allow a test of the overhead controllability of the time-slice scheduler because a reconfiguration is restricted differently for CPU usage by these parameters. Moreover, the computing and reconfiguration loads are set differently for C-DEDSS and F-DEDSS, and therefore, the throughput and response time of C-DEDSS is different from those of F-DEDSS. However, where results reproducible, we would find the same changing patterns for a particular strategy in both C-DEDSS and F-DEDSS. In addition, the target system's performance was reported lower than that of the original system because the computing load of the target system was set higher, reflecting the reconfiguration scenario that compression/decompression components are added. The workload is also set differently for each TC by varying the number of concurrent clients and the frequency for submitting requests from each client. However, computing and reconfiguration load and workload from clients are set the same for all strategies in a single TC for sound comparison. In addition, each reconfiguration strategy is tested separately but started at the same time point indicated by symbol S (only Figs. 5–8). Different strategies finish at different time points. The longest strategy is indicated by symbol E (only Figs. 5–8).

Common to all TCs, the results have demonstrated that:

1. the running system's QoS, measured either by throughput or response time (milliseconds), goes through three phases: installation of new components, transformation of the structure, and removal of the old components, for each strategy;
2. each strategy's impact on the running system's QoS at each phase is clearly identifiable;
3. the results of each TC are reproducible on both C-DEDSS (on a two-processor machine) and F-DEDSS (on a single-processor machine);
4. each strategy behaves similarly in each TC on both C-DEDSS and F-DEDSS;
5. there is a positive correlation between throughput and response time. If strategy$_1$ is better than strategy$_2$ in throughput, strategy$_1$ must be better than strategy$_2$ in response time and vice versa.

The above five features confirm the effectiveness and reliability of the evaluation results.

8.3. Results analysis

For strategy *avl-cpt*, the use of quiescence imposed the sharpest impact. The running system's throughput experienced down-to-zero decline, and the system's responsiveness experienced the longest delay for both CPU saturation and non-saturation on both C-DEDSS and F-DEDSS. All other strategies exhibited better performance than *avl-cpt* due to removal of the blocking operation.

For strategies **-cpt*, competition for resources came from the installation phase and removal phase. A full competition capacity of reconfiguration caused a significant decline in throughput and responsiveness.

For strategies *con-cpt* and *sel-cpt*, stateless equivalence promoted the maintenance of QoS in terms of virtual instantaneity for state transfer. By shortening the period of throughput or responsiveness decline, *sel-cpt* exhibited better performance than *con-cpt*, regardless of the problem scale of state transfer.

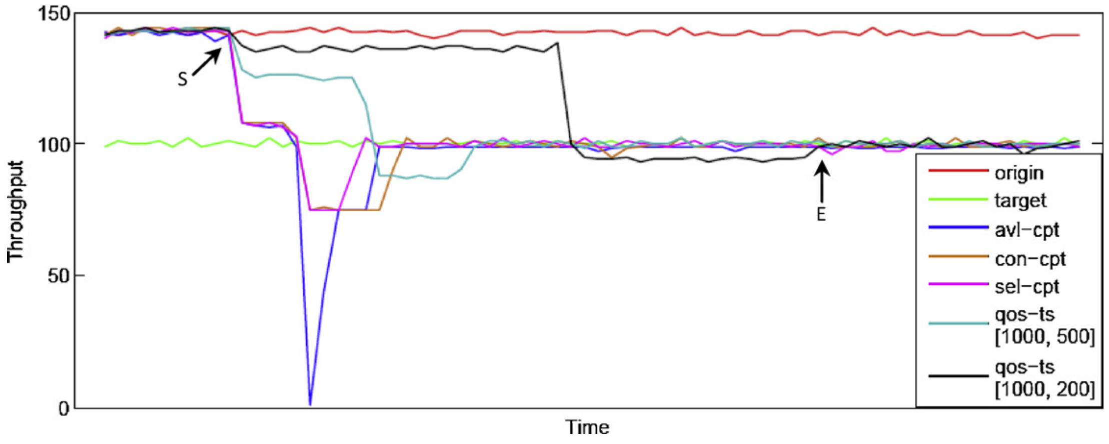

Fig. 5. The running system's QoS of C-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [saturation, throughput].

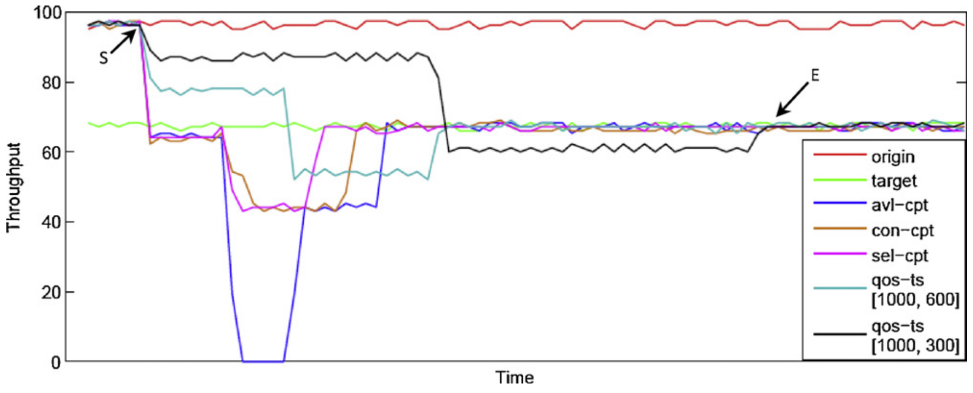

Fig. 6. The running system's QoS of F-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [saturation, throughput].

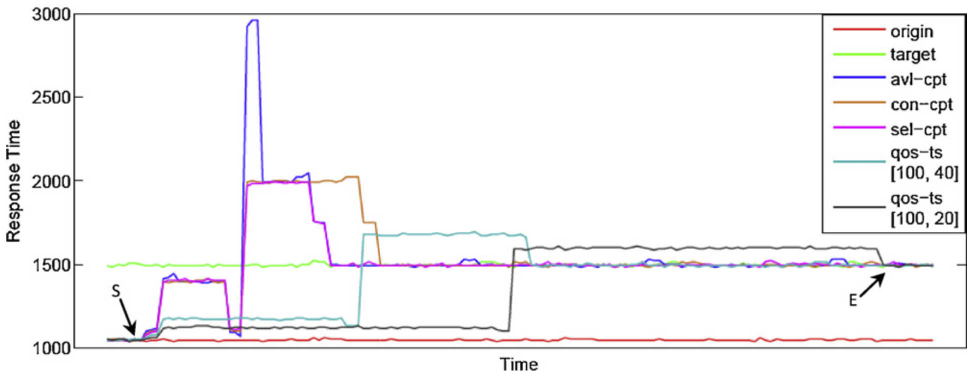

Fig. 7. The running system's QoS of C-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [saturation, response time].

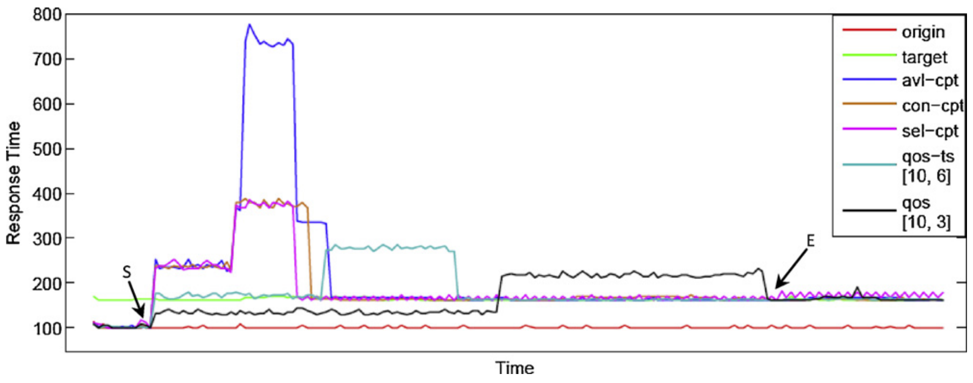

Fig. 8. The running system's QoS of F-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [saturation, response time].

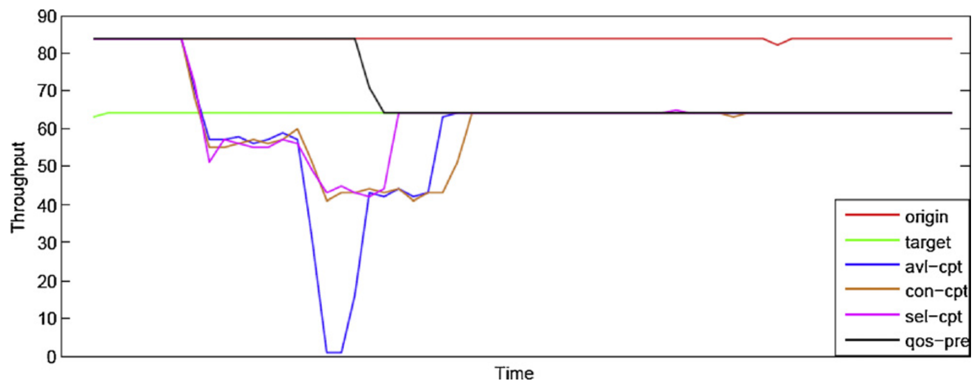

Fig. 9. The running system's QoS of C-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [non-saturation, throughput].

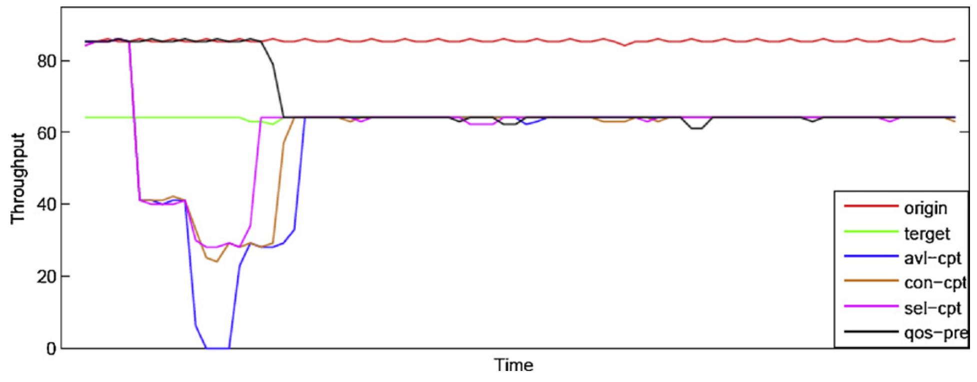

Fig. 10. The running system's QoS of F-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [non-saturation, throughput].

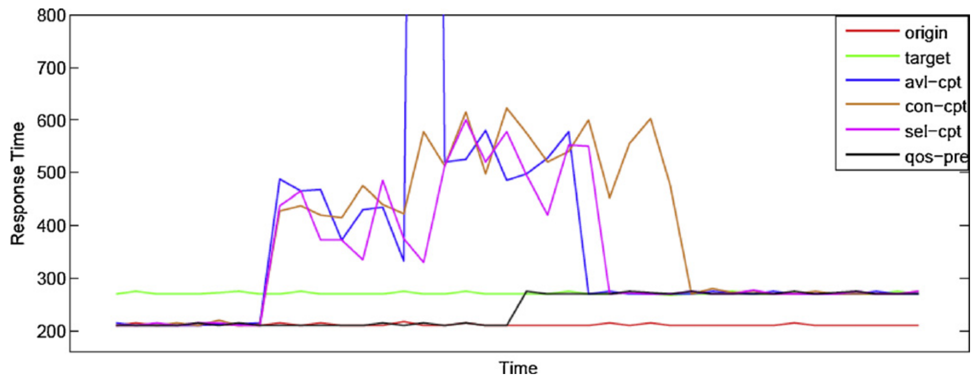

Fig. 11. The running system's QoS of C-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [non-saturation, response time].

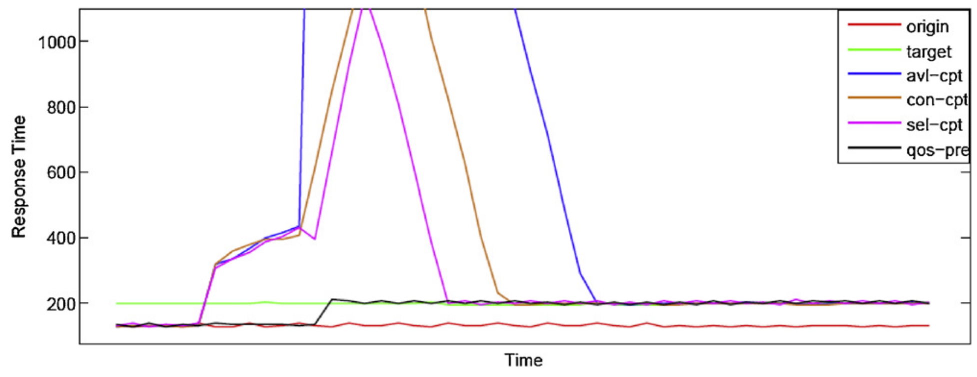

Fig. 12. The running system's QoS of F-DEDSS under different reconfiguration strategies going through the reconfiguration period for TC: [non-saturation, response time].

For strategies *qos-ts** (only Figs. 5–8), time-slice scheduling provided a full controllability over reconfiguration overhead when the CPU was always saturated by ongoing transactions. This can be shown as follows. First, *qos-ts** caused a smaller throughput decline or a shorter delay of response time than other strategies. Second, a shorter runnable time-slot strategy, such as *qos-ts [1000,300]*, had better performance than a longer runnable time-slot strategy, such as *qos-ts [1000,600]*, in terms of both throughput and responsiveness.

For strategy: *qos-pre* (only Figs. 9–12), pre-emptive scheduling removed the impact of reconfiguration on the running system's QoS. Both throughput and response time changed from the original to the target line directly without any decline or fluctuation.

8.4. Explanation of QoS impacts

We have demonstrated the reproducibility of the evaluation results for two structures: C-DEDSS and F-DEDSS on two machines and a positive correlation for a reconfiguration strategy between throughput and response time. These features confirm that the results are reliable, and the following explanations make sense.

If avoidance of downtime is the major concern and rapid reconfiguration is preferred, quiescence or tranquility is more applicable. The tradeoff is a sharp QoS deterioration during the reconfiguration period.

If QoS is the major concern in terms of stable throughput or response time, DVM must be applied to ensure service continuity. However, QoS maintenance only by DVM is just a logical but not physical control due to the following reasons.

State transfer could cause significant QoS decline, but stateless equivalence ensures that QoS is not affected by state transfer in terms of state's size or transfer occasion.

If predetermined QoS in terms of throughput and response time must be maintained, control of reconfiguration overhead must be applied in consideration of CPU usage. Pre-emptive scheduling makes use of free CPU time and enables reconfiguration to have no impact on QoS for CPU non-saturation case. Time-slice scheduling restricts reconfiguration for CPU usage and therefore ensures a certain CPU usage for the ongoing transactions. Consequently, a predetermined QoS can be maintained. However, the tradeoff is a longer reconfiguration period.

In conclusion, the QoS strength of all strategies can be sorted in an ascending order: *avl-cpt* (availability), *con-cpt* (continuity), *sel-cpt* (continuity plus stateless equivalence), and *qos-ts* (QoS assurance; CPU saturation) or *qos-pre* (QoS assurance; CPU non-saturation).

9. Conclusions and future work

The conducted evaluation is comprehensive not only in terms of qualitative modeling and analysis but also in empirical terms of quantitative comparison of the QoS assurance capabilities of available dynamic reconfiguration approaches.

The legitimacy of the proposed evaluation benchmark comes from two aspects. First, DEDSS accommodates a rich set of QoS problems of interest. Second, conceptual DEDSS may be translated into different architectural implementations for extensive and sound experimentation.

The legitimacy of the evaluation method can be confirmed by two facts. First, reconfiguration strategies are an abstraction and realization of available reconfiguration approaches, and enable a sound comparison of them in a unified evaluation context. Second, the evaluation results are effective, since:

1. the evaluation was conducted on two different architectural prototypes (F-DEDSS and C-DEDSS) of the conceptual evaluation benchmark: DEDSS. The evaluation results are reproducible;
2. the evaluation was conducted on two different hardware platforms with different threading policies to reflect onto the application layers. The tests show that the evaluation results are reproducible;
3. the positive correlation between throughput and response time is demonstrated for each reconfiguration strategy, confirming the reliability of the results.

Three steps towards QoS assurance have been confirmed. The first step is DVM, which ensures the logical continuity of a running system's service in the face of reconfiguration. The second step is stateless equivalence, which avoids the unintentional blocking to a running system's workflow by state transfer. The third step is overhead control, which restrains a reconfiguration to utilize free resources or predetermined amount of resources.

Global consistency is addressed by all existing approaches. However, most have to rely on quiescence, which results in significant QoS decline. Some approaches have addressed service continuity, but they have ignored the unintentional blocking problem and overhead control. Consequently, their QoS assurance capability is limited. The new approaches considered in this article have provided a more comprehensive treatment of the spectrum of QoS problems.

One issue for future investigation is the exploration of QoS assurance in the face of component migration in distributed environments. State migration will be an essential feature in support of QoS assurance of reconfiguration in such environments. The challenge will be to reduce the impact on QoS by state migration from one physical machine to another. It is expected that remote referencing will provide some control of QoS for state transfer. However, appropriate timing needs to be studied for physical transfer of the state. Another issue relates to modeling adaptive overhead control when CPU usage varies between non-saturation and saturation. Once these issues have been studied, a more comprehensive evaluation be possible.

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 proof-reading of this article.

References

Ajmani, S., Liskov, B., Shrira, L., 2006. Modular software upgrades for distributed systems. In: Proc. of 20th European Conf. on Object-Oriented Programming. Springer, pp. 452–476.

Almeida, J.P.A., Sinderena, M.v., Piresa, L.F., Wegdama, M., 2004. Platform-independent dynamic reconfiguration of distributed applications. In: Proc. of 10th IEEE International Workshop on Future Trends of Distributed Computing Systems. IEEE Computer Society Press, pp. 286–291.

Arshad, N., Heimbigner, D., Wolf, A.L., 2003. Deployment and dynamic reconfiguration planning for distributed software systems. In: Proc. of 15th IEEE International Conf. on Tools with Artificial Intelligence ,. IEEE Computer Society Press, pp. 39–46.

Bidan, C., Issarny, V., Saridakis, T., Zarras, A., 1998. A dynamic reconfiguration service for CORBA. In: Proc. of 4th International Conf. on Configurable Distributed Systems ,. IEEE Computer Society Press, pp. 35–42.

Cook, J.E., Dage, J.A., 1999. Highly reliable upgrading of components. In: Proc. of 1999 International Conf. on Software Engineering ,. IEEE Computer Society Press, pp. 203–212.

Dowling, J., Cahill, V., 2001a. The K-component architecture meta-model for self-adaptive software. In: Proc. of 3rd International Conf. on Metalevel Architectures and Separation of Crosscutting Concerns ,. Springer, pp. 81–88.

Dowling, J., Cahill, V., 2001b. Dynamic software evolution and the K-component model. In: Proc. of OOPSLA 2001 Workshop on Software Evolution ,. ACM Press.

Evans, H., 2004. DRASTIC and GRUMPS: the design and implementation of two run-time evolution frameworks. IEE Proceedings Software 151 (2), 30–48.

Feng, T., Maletic, J., 2006. Applying dynamic change impact analysis in component-based architecture design. In: Proc. of the 7th ACIS International Conf. on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'06) ,. IEEE Computer Society Press, pp. 43–48.

Fung, K., Low, G., 2009. Methodology evaluation framework for dynamic evolution in composition-based distributed applications. Journal of Systems and Software 82 (12), 1950–1965.

Hicks, M., Moore, J.T., Nettles, S., 2005. Dynamic software updating. ACM Transactions on Programming Languages and Systems 27 (6), 1049–1096.

Hillman, J., Warren, I., 2004a. An open framework for dynamic reconfiguration. In: Proc. of 26th International Conf. on Software Engineering ,. IEEE Computer Society Press, pp. 594–603.

Hillman, J., Warren, I., 2004b. Quantitative analysis of dynamic reconfiguration algorithms. In: Proc. of International Conf. on Design, Analysis and Simulation of Distributed Systems ,. The Society for Modeling & Simulation International (SCS).

Janssens, N., 2006. Dynamic Software Reconfiguration in Programmable Networks. PhD Thesis, Katholieke Universiteit Leuven, Belgium.

Janssens, N., Truyen, E., Sanen, F., Joosen, W., 2007. Adding dynamic reconfiguration support to JBoss AOP. In: Proc. of 1st Workshop on Middleware–Application Interaction ,. ACM Press, pp. 1–8.

Kim, D.K., Bohner, S., 2008. Dynamic reconfiguration for Java applications using AOP. In: Proc. of IEEE SoutheastCon ,. IEEE Press, pp. 210–215.

Kramer, J., Magee, J., 1990. The evolving philosophers problem: dynamic change management. IEEE Transactions on Software Engineering 16 (11), 1293–1306.

Lee, E.A., Parks, T.M., 1995. Dataflow process networks. Proceedings of the IEEE 83 (5), 773–801.

Leger, M., Ledoux, T., Coupaye, T., 2010. Reliable Dynamic Reconfigurations in A Reflective Component Model. Lecture Notes in Computer Science, vol. 6092. Springer, pp. 74–92.

Li, W., 2009. DynaQoS-RDF: a best effort for QoS assurance of dynamic reconfiguration of dataflow systems. Journal of Software Maintenance and Evolution: Research and Practice 21 (1), 19–48.

Mitchell, S., Naguib, H., Coulouris, G., Kindberg, T., 1999. A QoS support framework for dynamically reconfigurable multimedia applications. In: Proc. of International Conf. on Distributed Applications and Interoperable Systems ,. Kluwer, pp. 17–30.

Morrison, R., Balasubramaniam, D., Kirby, G., Mickan, K., Warboys, B., Greenwood, R.M., Robertson, I., Snowdon, B., 2007. A framework for supporting dynamic system co-evolution. Automated Software Engineering 14 (3), 261–292.

Moser, L.E., Melliar-Smith, P.M., Tewksbury, L.A., 2002. Online upgrades become standard. In: Proc. of 26th Annual International Computer Software and Applications Conf. ,. IEEE Computer Society Press, pp. 982–988.

Najjar, W.A., Lee, E.A., Gao, G.R., 1999. Advances in the dataflow computational model. Parallel Computing 25 (13–14), 1907–1929.

Patterson, D., 2002. Recovery oriented computing: a new research agenda for a new century, keynote. In: 8th International Symposium on High-Performance Computer Architecture (HPCA 8). <http://www.cs.berkeley.edu/~pattrsn/talks/keynote.html> (accessed 31.03.11).

Palma, N.D., Laumay, P., Bellissard, L., 2001. Ensuring dynamic reconfiguration consistency. In: Proc. of 6th International Workshop on Component-Oriented Programming (WCOP 2001).

Rasche, A., Polze, A., 2008. ReDAC – dynamic reconfiguration of distributed component-based applications with cyclic dependencies. In: Proc. of 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing ,. IEEE Computer Society Press, pp. 322–330.

Senivongse, T., 1999. Enabling flexible cross-version interoperability for distributed services. In: Proc. of International Symposium on Distributed Objects and Applications ,. IEEE Press, pp. 201–210.

Solarski, M., Meling, H., 2002. Towards upgrading actively replicated servers on-the-fly. In: Proc. of 26th Annual International Computer Software and Applications Conf. ,. IEEE Computer Society Press, pp. 1038–1043.

Soria, C., Munoz, D., Perez, J., Carsi, J., 2008. Managing dynamic evolution of architectural types. Lecture Notes in Computer Science, vol. 5292. Springer, pp. 282–289.

Soria, C., Munoz, D., Perez, J., Carsi, J., 2009. A reflective approach for supporting the dynamic evolution of component types. In: Proc. of 14th IEEE International Conf. on Engineering of Complex Computer Systems ,. IEEE Computer Society Press, pp. 301–310.

Surajbali, B., Grace, P., Coulson, G., 2010. Preserving dynamic reconfiguration consistency in aspect oriented middleware. In: Proc. of 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software ,. Hasso Plattner Institut, pp. 33–40.

Tewksbury, L.A., Moser, L.E., Melliar-Smith, P.M., 2001. Live upgrades of CORBA applications using object replication. In: Proc. of IEEE International Conf. on Software Maintenance ,. IEEE Computer Society Press, pp. 488–497.

Truyen, E., Janssens, N., Sanen, F., Joosen, W., 2008. Support for distributed adaptations in aspect-oriented middleware. In: Proc. of 7th International Conf. on Aspect-Oriented Software Development ,. ACM Press, pp. 120–131.

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

Warren, I., 2000. A Model for Dynamic Configuration which Preserves Application Integrity, PhD Thesis, Lancaster University, UK.

Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T., 2006. An automated formal approach to managing dynamic reconfiguration. In: Proc. of 21st IEEE International Conf. on Automated Software Engineering ,. IEEE Computer Society Press, pp. 37–46.

Zhang, J., Yang, Z., Cheng, B., McKinley, P., 2004. Adding safeness to dynamic adaptation techniques. In: Proc. of ICSE 2004 Workshop on Architecting Dependable Systems ,. ACM press, pp. 17–21.

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