Evaluation of composite object replication schemes for dependable server applications

Panagiotis Katsaros Nantia Iakovidou Theodoros Soldatos

Department of Informatics

Aristotle University of Thessaloniki

54124 Thessaloniki, Greece

tel.: +30-2310-998532, fax.: +30-2310-998419

{ katsaros, niakovid, thsoldat}@csd.auth.gr

Abstract

Object oriented dependable server applications often rely on fault tolerance schemes, which are comprised of different replication policies for the constituent objects (composite replication schemes). This paper introduces a simulation-based evaluation approach for quantifying the tradeoffs between fault-tolerance overhead and fault tolerance effectiveness in composite replication schemes. Compared to other evaluation approaches: (a) we do not use the well-known reliability blocks based simulation, but a hybrid reliability and system's traffic simulation and (b) we make a clear distinction between the measures used for the fault-affected service response times from those used for the fault-unaffected ones. The first mentioned feature allows taking into account additional concerns other than fault tolerance, like for example load balancing and multithreading. The second feature renders the proposed approach suitable for design studies that aim to determine either optimal replication properties for the constituent objects or Quality of Service (QoS) guarantees for the perceived service response times. We obtain results for a case system model, based on different assumptions on what happens when server-objects fail (loss scenarios). The presented results give insight in the design of composite method request-retry schemes with appropriate request timeouts.

KEYWORDS: Fault-tolerance, Performance, Quality of Service, Distributed objects, Simulation

1. Introduction

In dependable applications fault tolerance often grounds on the utilization of redundant processing for minimizing loss of computation in the presence of non-recurrent faults. Common sources of faults that do not recur after recovery are: insufficient memory, media failures, power outages, network failures and the non-determinism introduced either by distributed timers or the use of multithreading.

Fault tolerance schemes for object-oriented applications are possibly comprised of different replication policies for the constituent objects. We call them composite replication schemes. According to the recently published OMG FT-CORBA specification ([19]), the replication policy assigned to an object can be either active, warm passive or cold passive replication and is customized by a set of behavioral properties (number of replicas, checkpoint/state transfer interval, request-retry timeout etc) with values that are found to be appropriate for the used fault-detection setting.

In dependable server applications, where fault tolerance is attained via composite replication schemes, the perceived quality of service (QoS) is dominated by characteristic tradeoff concerns between fault tolerance overhead and fault tolerance effectiveness: (i) excessively frequent checkpoints (if any) result in performance degradation, while deficient checkpoints incur expensive recovery and (ii) excessively frequent invocation request-retry timeouts cause high overhead costs and do not improve fault-tolerance effectiveness.

The perceived QoS is also determined by the system's method call dependencies, because the callers are blocked until they get the invoked method replies.

A composite replication scheme may be combined with a request assignment strategy for load balancing to provide possibly agreed QoS guarantees. Checkpoints and state transfers between object replicas, as well as object replica recoveries affect the invoked requests' dispatching and for this reason influence the performance of the applied load balancing.

In this computational context, when an application has to conform to possibly agreed QoS guarantees regarding service response times, the published reliability blocks based evaluation techniques ([9]) are inadequate.

The current work proposes a hybrid reliability and system's traffic simulation: events taking place in different time scales - like transient faults, which by definition are rare events and service request arrivals that are not rare events - are simulated together in the same experiment. This allows taking into account complex interactions that are otherwise attributed to diverse design concerns (fault tolerance, load balancing and multithreading). Also, the proposed reliability and system's traffic simulation allows evaluation of measures other than reliability (response times, throughput etc).

In current work, we make a clear distinction between the measures used for the fault-affected service response times from those used for the fault-unaffected ones. The former class of response times takes into account the effects of recovery, i.e. the amount of computation lost due to rollbacks and/or object state transfers as a result of the occurred object faults. Their mean quantifies the achieved *fault-tolerance effectiveness*. The latter class of response times includes the vast majority of serviced requests. These are the requests that are not influenced by the occurred object faults (which by definition are rare events), but take into account the overhead costs due to checkpointing and object state transfers incurred by the applied composite replication scheme. Their mean characterizes the resulted *fault-tolerance performance*.

The perceived quality of service (QoS) and the associated guarantees are concerned with both classes of service requests. However, our aim is to not only provide yet another "working" evaluation approach: the separate quantification of fault tolerance performance and fault tolerance effectiveness gives insight into the most influential performance and effectiveness tradeoffs and the complex interactions due to the system's method call dependencies. As a consequence, the proposed evaluation approach may be the cornerstone of systematic QoS design methods as for example the one we introduced in [10].

In current paper we provide formal definitions of the so-called fault affected and fault unaffected service response times and we describe the prototype simulator that implements the proposed evaluation approach. We provide results for a case system model, based on different assumptions for what happens when server-objects fail (loss scenarios). The obtained results give insight in the design of composite service request-retry schemes with appropriate object-request timeouts.
Section 2 outlines some closely related evaluation approaches. Section 3 describes the adopted computational model, the assumed object fault assumptions and the developed prototype simulator. Section 4 provides a formal description of the proposed evaluation approach. Section 5 introduces the used case system model and summarizes the obtained fault-tolerance performance and fault-tolerance effectiveness results. Finally, we conclude with a discussion on the perspectives of the proposed evaluation approach.

2. Related work

Analytic performance models for software replication have focused on the evaluation of process-based replication ([6]) and are not appropriate for applications where fault tolerance involves different replication policies for the constituent objects.
In [3], the authors introduce an analytic model termed as Fault-Tolerant Layered Queueing Network (FTLQN), for efficiently predicting the performability in two specific classes of fault-tolerant client-server systems. Performability estimation is based on efficient layered queueing network models that are combined with AND-OR graph analysis.
Simulation-based performance evaluation of fault tolerance has been previously reported in [23] and [22]. The first mentioned article refers to the evaluation of process-based message logging, checkpointing and recovery schemes, whereas the second one introduces an approach for relative performance comparison of different checkpointing and recovery protocols. None is focused on the evaluation of replication schemes comprised of possibly different replication policies for the constituent objects.

Worth to mention is the work published in [14], where the authors propose a hybrid mathematical programming and analytic evaluation algorithm for a trade-off problem that is not directly related to fault-tolerance: to determine process replication or threading levels, such as to avoid unnecessary queuing delays for request senders or unnecessary high consumption of memory.

3. The prototype simulator

The prototype simulator we introduce in this section reflects the need to keep the set of model's parameters and the set of simulated event types as small as possible and at the same time to provide credible results that give insight into the most influential fault tolerance tradeoff concerns. The models' level of abstraction is perceived by the set of parameters used (see the case study in Section 5) and takes into account the hardware as well as the software resource contention caused by the applied composite replication scheme. The credibility of the produced results is ensured by the applied simulation output analysis (section 5). The resulted accuracy, as in any simulation study, depends on the availability of appropriate resource consumption and fault occurrence data.

Although a detailed description of the developed simulator is beyond the scope of this paper, we proceed to the description of the functionality that we believe is necessary to introduce the proposed evaluation approach. The prototype tool is a standalone application developed in C++, which features an extensible object-oriented design and does not make use of special components and libraries. The simulator's description refers to the adopted computational model, the simulated object fault models, the behavioral specifications of the implemented object replication policies and the assumed fault detection setting.

3.1 Computational model

Object replication is studied in the context of the OMG Core Object Model ([20]) including the assumptions adopted in the OMG FT-CORBA specification ([21]).

An object is characterized by its distinct object identity, which is immutable, persists for as long as the object exists and is independent of the object’s properties or behavior. Each object owns or does not own a state and a set of methods.

Each method has a signature that includes the method’s name, the set of parameters and the set of results. A *method invocation*, also called *request*, can list some parameters on behalf of a requester (client) and can cause the method to return results. The methods of an object constitute the only way to change its state.

We only consider the use of *synchronous method invocations*: the object requiring the execution of the invoked method (requester) stops executing and waits for the invoked execution to terminate and the reply to return. Upon reception of the *reply*, the sender resumes. A computation invoked in a server application object (*service object*) will be called *service request* and is possible to involve one or more synchronous and possibly *nested requests* to other objects.

We adopt the *“at-most-once” invocation semantics* of the OMG Core Object Model, which means that each request is executed at most once. Duplicate method invocations due to replication or request-retry are detected and suppressed.

In OMG FT-CORBA, fault-tolerance is based on the creation and management of multiple object replicas as a single *object group*. The client objects invoke methods on the server object group and one or more members of the server group execute the methods and return their responses to the clients, just like a conventional object.

We do not place any assumptions about the network topology. For the protocols making up the interprocess communication we adopt the OMG FT-CORBA assumption that they provide *totally ordered delivery of requests* to the replicas of each object.

Strong replica consistency (OMG FT-CORBA): even in the presence of faults, as members of an object group execute a sequence of methods invoked on the object group, its behavior is logically equivalent to that of a single fault-free object processing the same sequence of method invocations. For each object, strong replica consistency retains an appropriate context that depends on the object group's *replication policy* (*active, warm passive* or *cold passive*).

Although the proposed evaluation approach is open to take into account different *multithreading* possibilities, the prototype simulator implements the thread-per-object concurrency approach ([26]), where a single thread executes all invocations on an object replica.

We assume *deterministic behavior* of the underlying operating system. Ordering of dispatched method invocations at the application programming level is not allowed. Finally, the application either does not make use of system calls returning processor-specific information or such calls are handled by an appropriate mechanism without introducing non-determinism.

Load balancing is used as a mean to show the potentiality of the proposed approach to take into account additional design concerns, other than fault tolerance. We do not distinguish between centralized or distributed load balancing. Load information (if applicable) is provided by the underlying middleware infrastructure, which also works out request assignment to the available service objects in a transparent and fault tolerant manner (as in [27] and [13]).

State consistency management across object groups, where service requests are assigned, violates the principle of application transparency, due to its dependence on the application of interest. Thus, we are restricted to load balancing strategies with no consistency management costs: the synthetic workload scenario used in our case study assumes request assignment to the stateless service objects on a round-robin basis.

3.2 Simulated object fault models

We model faults that do not recur after recovery and are eventually manifested as *transient object faults*. Application objects conform to the *fail-stop model* ([25]), which means that they fail by crashing, without emission of spurious messages. Commission faults as for example Byzantine faults, where an object generates incorrect results are not addressed.

We allow modeling of network faults, where the client does not detect the fault and receives no reply. However, as in the related OMG FT-CORBA specification, we exclude network-partitioning faults that separate the hosts of the system into two or more sets.

In general, when a service object fails, the already queued requests are not lost. All requests arriving while the object is down are queued. For the passively replicated objects that own a state we have also implemented the following types of *omission fault scenarios* (loss behavior):

a. When the object fails, no requests previously accepted in the queue are lost but all requests arriving while the object is down are lost.
b. When the object fails, the requests currently in service, if any, are lost and the requests arriving while the object is down are also lost.
c. At the instant the server object fails, the requests currently in service, if any, are lost and the requests arriving while the object is down are queued.
d. At the time the server object fails, the already queued requests are lost and the requests arriving while the object is down are also lost.
e. At the time the server object fails, the already queued requests are lost, but the requests arriving while the object is down are queued.

As in OMG FT-CORBA, the prototype simulator supports a *request-retry timeout mechanism* as a mean to mask network and recipient omission faults, where the client-side does not detect the problem and receives no reply. A request-retry incurs overhead processing for re-invoking the (possibly lost) request.

The prototype simulator allows: (i) to take into account different fault propagation scenarios, (ii) to use alternative object fault - repair distributions and (iii) to apply load-dependent fault models, like for example those used in [7].

3.3 Object replication

In *active replication* all the object group replicas execute each invocation independently, but in the same order (state N in Figure 1). Object replicas maintain exactly the same state and in case of a fault in one replica (state F in Figure 1), the simulated application continues with the results provided by the other replicas, without having to wait for fault detection and recovery (state R in Figure 1).

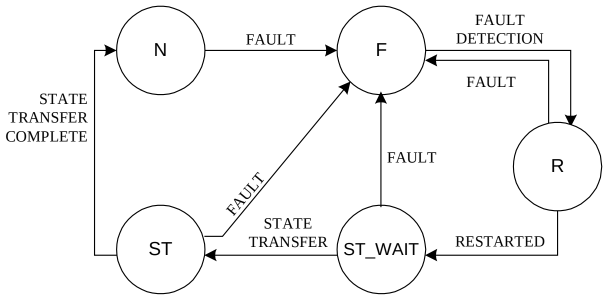
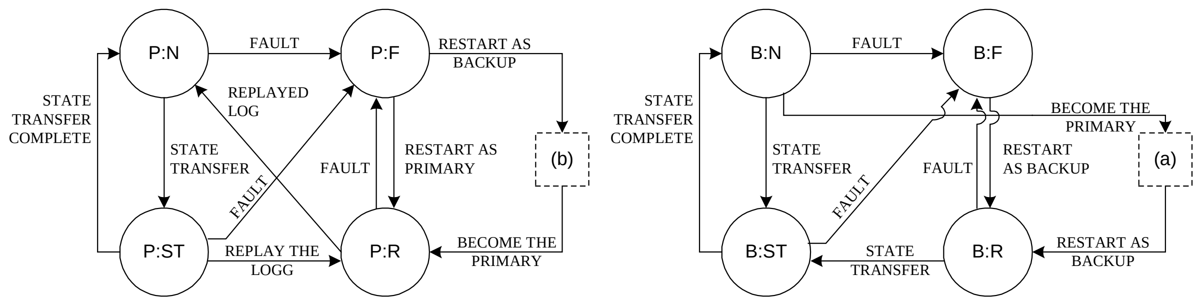


Figure 1 Object replica in an actively replicated object group

Strong replica consistency for active replication means that, at the end of each method invocation on the object group, all the group members have the same state. Each group member responds to all incoming requests, but duplicate requests/replies are detected and suppressed, thus delivering only a single request/reply to the destination object.

When an object replica is recovered, a state transfer (and checkpoint) from a live replica is accomplished (state ST in Figure 1). Since an object state transfer requires operational quiescence in participating replicas, the state transfer is postponed (state ST_WAIT in Figure 1) when all other replicas are in-between an invocation service. If, in the course of recovery or a state transfer, object replicas receive additional invocations, all of them are queued locally and subsequently applied to them.

Cold or *warm passive replication* assumes that during fault-free operation, only one member - the *primary* (Figure 2a) - of the object group executes the methods invoked on the group. The state of the primary and the sequence of the invoked methods are recorded in a *message log*, according to the applied checkpoint properties.



(a) primary object replica (b) backup object replica

Figure 2 Warm passive replication with one backup object

A checkpoint/state transfer is postponed when the primary is in-between an invocation service or it happens to be blocked, waiting for a response. In the course of a checkpoint or a state transfer activity, new invocations may be received, but they cannot be processed, before the end of it.

Strong replica consistency implies that at the end of a checkpoint/state transfer (transitions P:ST→P:N and B:ST→B:N in Figure 2), all replicas own or have access to the same state. In the presence of a fault (state P:F in Figure 2a), a backup replica is promoted to be the new primary (transition B:N→P:R). The state of the new primary is restored to the state of the old one, by reloading the last saved checkpoint and subsequently reapplying the request messages that have been recorded in the message log. This implies that a client can re-invoke a request on a server and receive a reply to that request, but as we already noted without risk that the method will be executed more than once.

In cold passive replication, the backup replicas are not activated. When the current primary fails, a new one is selected and then activated. In warm passive replication, the backups have been already activated and their states are continuously synchronized with the primary replica’s state, according to the specified frequency of state transfers.

Resource consumption for checkpoints and state transfers depends on the object state size and on the processing speed of the slowest participating object replica.

3.4 Fault detection

In fault tolerant systems, the fault detector that monitors an application object is usually located, for efficiency, on the same host as that of the object. A global fault detector that is replicated for fault tolerance monitors the local fault detectors. It has been found ([6]) that fault monitoring causes an approximate 5% increase, in the processor (of a Pentium-II based 200+ MHz machine) utilization, for about 500 milliseconds.

In our prototype simulator, each object is periodically checked, according to a specified time interval that represents the sum of the fault monitoring interval plus the time allowed for subsequent response from the object, to determine whether it is faulty. The forenamed overhead processing is included in the system model's workload and is raised proportionately to the used fault-monitoring interval.

4. Fault tolerance performance and fault tolerance effectiveness

A distributed system is composed of multiple objects o_1, o_2, . . ., o_n. Each object o_i, $1 \le i \le n$ is replicated either according to the active replication policy of Figure 1 with k object replicas, $o_i^{ar_1}, o_i^{ar_2}, ..., o_i^{ar_k}$ or alternatively, according to the passive replication policy of Figure 2, with one primary replica o_i^{prim} and one backup o_i^{back}.

An object's methods $op_l^{o_i}$, $1 \le l \le$ #(methods of o_i) may be synchronously invoked by remote method invocations. A method invocation is represented by an ordered pair of a request and a reply message ($rq(op_l^{o_i})$, $rp(op_l^{o_i})$), for simplicity denoted ($rq_l^{o_i}$, $rp_l^{o_i}$).

On receipt of a $rq_l^{o_i}$ the request is queued in the queues of all replicas o_i^r, denoted by $Q(o_i^r)$, where $r \in \{ar_j \mid 1 \le j \le \#(o_i^r)\} \cup \{prim\}$. Each o_i^r is placed on a separate object server and queued requests are served in FIFO ordering under a single thread of control. However, the proposed approach can also be extended for application in other multithreading cases. If $r = prim$, executed requests $rq_l^{o_i}$ are appended to a *local log queue* log_i.

Let $op^{o_i}_{l,p}$ $(p \in \aleph)$ denote a method invocation instance of $op^{o_i}_{l}$. Each $op^{o_i}_{l,p}$ may further invoke another method instance $op^{o_t}_{s,p}$, with $1 \leq s \leq$ #(methods of o_t), $1 \leq t \leq n$, $t \neq i$ and this is the case of a nested invocation, where $op^{o_i}_{l,p}$ is blocked up to the reception of $rp^{o_t}_{s,p}$. Access to a simulated object's state is performed by primitive read/write messages that induce computational resource consumption, but they are not included in the methods' message sequence specifications.

A *message sequence specification* for $op^{o_i}_{l}$, $1 \leq l \leq$ #(methods of o_i) is given as a total order relation < over the set

$$MsgSeq(op^{o_i}_{l}) = \{ op^{o_t}_{s} \mid 1 \leq s \leq \#(\text{methods of } o_t),\ 1 \leq t \leq n \text{ and } t \neq i\}$$

of nested invocations generated by $op^{o_i}_{l}$. This set is empty when $op^{o_i}_{l}$ causes exclusively read/write operations on the state of the simulated o_i and no nested invocations. A method $op^{o_{t1}}_{s1}$ is referred to as *preceding another method* $op^{o_{t2}}_{s2}$ ($op^{o_{t1}}_{s1} < op^{o_{t2}}_{s2}$) if and only if $rq^{o_{t2}}_{s2}$ may be sent only after $rp^{o_{t1}}_{s1}$ is already received.

A *global state S* of the system consists of all *local log queues log*$_i$ and local queues

$$Q(o^r_i) \text{ with } r \in \{ar_m \mid 1 \leq m \leq \#(o^r_i)\} \cup \{prim, back\} \text{ and } 1 \leq i \leq n$$

plus the values of $3\sum_{i=1}^{n} \#(o^r_i)$ additional boolean variables that are defined as follows:

- $crash^r_i$: these variables are initially false and become true at the time that o^r_i becomes faulty.
- $failed^r_i$: these variables are initially false and become true when the system detects that $crash^r_i$ = true.
- $recovered^r_i$: these variables are initially true to indicate that o^{prim}_i reflects the object state of having executed all $rq^{o_i}_{s,q} \in log_i$, with $1 \leq s \leq$ #(methods of o_i). In any other case $recovered^r_i$ is false.

An initial global state S_0 is the global state in which all $Q(o^r_i)$ and log_i do not have queued requests (empty) and the additional boolean variables are set to their initial values. An *event* is an action that changes the global state of the system from S to S'. We will use the notation $e(S) = S'$ to denote that e occurs in global state S and results in global state S'.

We specify *chk_init*, *send*, *timeout*, *respond*, *crash*, *failed*, *restart* and *chk_end* event types by the following notation:

- *chk_init*(o_i^{r1}, o_i^{r2}) denotes the event whereby a *checkpoint/state transfer request* $c_i^{r1,r2}$ between o_i^{r1} and o_i^{r2} is placed second in the order of queues $Q(o_i^{r1})$ and $Q(o_i^{r2})$ (or at the head of them in case of empty queue(s)).
- *send*($o_i^{r1}, o_j, rq_{l,p}^{o_j}$, *nf*) denotes the event whereby o_i^{r1} sends $rq_{l,p}^{o_j}$ to all queues

  $$Q(o_j^{r2}),\ r_2 \in \{ar_m \mid 1 \le m \le \#(o_j^r)\} \cup \{prim\},$$

  where $p \in \aleph$ and $op_l^{o_j}$ is a member of the message sequence of the request at the head of $Q(o_i^{r1})$. In the absence of network fault (*nf* = false) a send event changes the local queue(s) $Q(o_j^{r2})$ if $rq_{l,p}^{o_j} \notin Q(o_j^{r2})$ and $crash_j^{r2}$ is false, by appending the request message $rq_{l,p}^{o_j}$. If $rq_{l,p}^{o_j} \notin Q(o_j^{r2})$ and $crash_j^{r2}$ is true then $Q(o_j^{r2})$ changes or does not change depending on the applied omission fault scenario (section 3.2). Finally, the event *timeout*($o_i^{r1}, rq_{l,p}^{o_j}$) is scheduled to occur, if it is required by the applied request-retry policy.
- *timeout*($o_i^{r1}, rq_{l,p}^{o_j}$) denotes the event whereby a request-retry timeout for $rq_{l,p}^{o_j}$ occurs. The event *send*($o_i^{r1}, o_j, rq_{l,p}^{o_j}$) takes place and a new *timeout*($o_i^{r1}, rq_{l,p}^{o_j}$) is then scheduled.
- *resp*($o_i^{r1}, o_j^{r2}, rp_{l,p}^{o_j}$, *nf*) denotes the event whereby o_j^{r2} responds with $rp_{l,p}^{o_j}$ to o_i^{r1} for some $r_2 \in \{ar_m \mid 1 \le m \le \#(o_j^r)\} \cup \{prim\}$. In the absence of network fault (*nf* = false) scheduled *timeout*($o_i^r, rq_{l,p}^{o_j}$) events (if any) are canceled for all $r \in \{ar_m \mid 1 \le m \le \#(o_i^r)\} \cup \{prim\}$ and not only for r_1. The local queue $Q(o_j^{r2})$ changes by removing $rq_{l,p}^{o_j}$ from the head and if $r_2 = prim$ then $rq_{l,p}^{o_j}$ is appended to log_j. If the next request is some $rq_{s,q}^{o_j}$, then o_j^{r2} proceeds to executing $rq_{s,q}^{o_j}$, based on $MsgSeq(op_s^{o_j})$. If the next

request is some $c_j^{r3,r2}$, then o_j^{r2} is blocked until $c_j^{r3,r2}$ is also placed at the head of $Q(o_j^{r3})$. A $chk\_end(c_j^{r3,r2})$ event is then scheduled to occur.

- $crash(o_i^r)$ denotes the event whereby $crash_i^r$ becomes true. This models the occurrence of a fault in o_i^r. Potential $resp(o_i^r, o_j^{r1}, rp_{l,p}^{o_j})$ and $timeout(o_i^r, rq_{l,p}^{o_j})$ events are ignored. If $r = prim$ then $Q(o_i^r)$ is changed according to the applied omission fault scenario (section 3.2).
- $failed(o_i^r)$ denotes the event whereby $failed_i^r$ becomes true. A recovery of o_i^r is then set up: in all replication cases, a $restart(o_i^r)$ event is scheduled.

  In the passive replication of Figure 2, if $r = prim$ and $crash_i^{back}$ is false, then $Q(o_i^r)$ is copied to $Q(o_i^{back})$, o_i^{back} becomes o_i^{prim} and r becomes *back*. This change results in

  $$failed_i^{prim} = crash_i^{prim} = \text{false and } failed_i^{back} = crash_i^{back} = \text{true}$$

  If $\#(rq_{s,q}^{o_i} \in log_i \mid 1 \le s \le \#(\text{methods of } o_i)) > 0$ then $recovered_i^{prim}$ becomes false and the variable keeps this value while o_i^{prim} has not yet replayed all $rq_{s,q}^{o_i} \in log_i$. When $recovered_i^{prim}$ changes to true, log_i is emptied.
- $restart(o_i^r)$ denotes the event whereby $crash_i^r$ and $failed_i^r$ become false.
- $chk\_end(c_i^{r1,r2})$ denotes the event whereby $c_i^{r1,r2}$ is removed from o_i^{r1} and o_i^{r2}. If r_1, r_2 are not *back*, they proceed to the execution of the next request $rq_{s,q}^{o_j}$ based on $MsgSeq(op_s^{o_j})$.

All forenamed events are atomic and each event affects only the relevant local queues and the relevant boolean variables. Thus, if $crash_i^r$ is false in the global state *S*, when $send(o_i^r, o_j, rq_{l,p}^{o_j})$ occurs, then $crash_i^r$ is also false in the resulting global state S'.

Definition 4.1: A *run of the system* is an infinite sequence of global states

$$run = (S_0, S_1, S_2, \ldots)$$

where S_0 is an initial global state and there exists a sequence of events $(e_0, e_1, e_2, \ldots)$ such that $\forall i \ge 0$, $e_i(S_i) = S_{i+1}$.

The *history of run* is the sequence of events $H_{run} = (e_0, e_1, e_2, \ldots)$ such that $\forall i \geq 0, e_i(S_i) = S_{i+1}$.

Table 1 Glossary of notation

o_i^r	object replica *r* of o_i	$crash_i^r$	there is an object fault at o_i^r
ar_j	active replica *j* of some object	$failed_i^r$	a fault at o_i^r has been detected
prim	the primary object replica of a passively replicated object	$re\text{cov}ered_i^r$	o_i^r reflects the object state of having executed all $rq_{s,q}^{o_i} \in log_i$
back	the backup object replica of a passively replicated object	$c_i^{r1,r2}$	request for checkpointing/state transfer between o_i^{r1} and o_i^{r2}
$op_{l,p}^{o_i}$	a method instance of $op_l^{o_i}$ with $1 \leq l \leq$ #(methods of o_i)	*run*	an infinite sequence of global states $S_0, S_1, \ldots$
$(rq_{l,p}^{o_i}, rp_{l,p}^{o_i})$	an ordered pair of request and reply messages that collectively represent the execution of $op_{l,p}^{o_i}$	H_{run}	history of *run*: a sequence of events $e_0, e_1, \ldots$ such that $\forall i \geq 0$, $e_i(S_i) = S_i + 1$
log_i	the log queue of a passively replicated o_i	$S_i \models \varphi$	predicate φ holds in global state S_i
$Q(o_i^r)$	the local queue of requests in o_i^r	*F_AFFECTED*($rq_{l,p}^{o_i}, o_j$)	
MsgSeq($op_l^{o_i}$)	the totally ordered set of nested invocations $op_s^{o_t}$ generated by $op_l^{o_i}$		boolean predicate indicating that $rq_{l,p}^{o_i}$ is affected by a fault at o_j

For any run *run*, H_{run} is uniquely determined and also, *run* can be constructed from the history H_{run} and the initial global state S_0.

As we already noted, we make a clear distinction between the measures used for the fault-affected service response times from those used for the fault-unaffected ones. The fault-affected service requests reflect the effects of recovery (fault-tolerance effectiveness). The fault-unaffected service requests include the overhead processing caused by the applied composite replication scheme and for this reason their mean characterizes the resulted fault-tolerance performance.

Definition 4.2: A synchronous request $rq_{l,p}^{o_j}$ to a passively replicated object *is affected by a fault at* o_j^{prim} in *run* and the boolean predicate *F_AFFECTED*($rq_{l,p}^{o_j}$, o_j) becomes true if and only if

H_{run}=(e_0; x; e_u= *send*(o_i^r , o_j , $rq_{l,p}^{o_j}$, false); y; e_v= *resp*(o_i^r , o_j^{prim} , $rp_{l,p}^{o_j}$, *nf*); w)

where x, y finite sequences of events with $e_u \notin x$, e_v is the first *resp* event with *nf* being either true or false and w an infinite sequence of events such that either:

i. $S_u \models \neg crash_j^{prim}$ and #($crash(o_j^{prim}) \in y$) > 0 or

ii. $S_u \models (crash_j^{prim} \vee \neg recovered_j^{prim})$ or

iii. $S_u \models (\exists\, rq_{s,q}^{o_j} \in Q(o_j^{prim})$, $1 \leq s \leq$ #(methods of o_j) such that $S_v \models F\_AFFECTED(rq_{s,q}^{o_j}, o_j)$)

If $S_v \models F\_AFFECTED(rq_{l,p}^{o_j}, o_j)$ and $S_v \models (\exists\, rq_{d,p}^{o_i}$ at the head of $Q(o_i^r)$, such that $op_l^{o_j} \in MsgSeq(op_d^{o_i})$, with $i \leq n$, $i \neq j$, $r \in \{ar_m \mid 1 \leq m \leq \#(o_i^r)\} \cup \{prim\}$, $1 \leq d \leq$ #(methods of o_i)), then $S_{v+1} \models F\_AFFECTED(rq_{d,p}^{o_i}, o_j)$.

The first condition reflects the case whereby $rq_{l,p}^{o_j}$ is sent to an operational o_j^{prim} and $crash_j^{prim}$ changes to true before the occurrence of the expected *resp* event. The second condition reflects the case whereby $rq_{l,p}^{o_j}$ is sent to a o_j^{prim}, that is not yet operational as a result of a *crash*(o_j^r) event. The third condition reflects the case whereby $rq_{l,p}^{o_j}$ is queued behind a $rq_{s,q}^{o_j}$ that is eventually affected by a *crash*(o_j^{prim}) event. If $rq_{l,p}^{o_j}$ is a nested invocation generated by $rq_{d,p}^{o_i}$ at o_i^r, then $rq_{d,p}^{o_i}$ is also affected by the *crash*(o_j^{prim}) event.

Definition 4.3: A synchronous request $rq_{l,p}^{o_j}$ to an actively replicated object *is affected by a fault at some* o_j^{r1} in *run* and the boolean predicate *F_AFFECTED*($rq_{l,p}^{o_j}$, o_j) becomes true if and only if

H_{run}=(e_0; x; e_u= *send*(o_i^r , o_j , $rq_{l,p}^{o_j}$, false); y; e_v= *resp*(o_i^r , o_j^{r2} , $rp_{l,p}^{o_j}$, *nf*); w)

where x, y finite sequences of events with $e_u \notin x$, e_v is the first *resp* event with $r_2 \in \{ar_m \mid 1 \leq m \leq \#(o_j^r)\}$ and *nf* being either true or false and w an infinite sequence of events such that either:

i. $\#(chk\_end(c_j^{r1,r2}) \in y) > 0$ and $r_1 \neq r_2$ or

ii. $\#(\text{crash}(o_j^{r1}) \in y) > 0$ with $r_2 = r_1$ or

iii. $S_u \models (\exists\, rq_{s,q}^{o_j}$ in all $Q(o_j^{r1})$, $r_1 \in \{ar_m \mid 1 \leq m \leq \#(o_j^r)\}$, $1 \leq s \leq \#(\text{methods of } o_j)$, such that $S_v \models F\_AFFECTED(rq_{s,q}^{o_j}, o_j))$

If $S_v \models F\_AFFECTED(rq_{l,p}^{o_j}, o_j)$ and $S_v \models (\exists\, rq_{d,p}^{o_i}$ at the head of $Q(o_i^r)$, such that $op_l^{o_j} \in MsgSeq(op_d^{o_i})$, with $i \leq n$, $i \neq j$, $r \in \{ar_m \mid 1 \leq m \leq \#(o_i^r)\} \cup \{prim\}$, $1 \leq d \leq \#(\text{methods of } o_i))$, then $S_{v+1} \models F\_AFFECTED(rq_{d,p}^{o_i}, o_j)$.

The first condition reflects the case whereby $rq_{l,p}^{o_j}$ has been blocked, in order to realize a state transfer for a replica recovery. The second condition reflects the case whereby o_j^{r1} is the first replica involved in a $resp(o_i^r, o_j^{r1}, rp_{l,p}^{o_j}, nf)$ event, but has previously become faulty in y. The third condition reflects the case whereby $rq_{l,p}^{o_j}$ is queued behind another request $rq_{s,q}^{o_j}$ that is eventually affected by a fault at some o_j^{r1}.

Definition 4.4: A synchronous request $rq_{l,p}^{o_j}$ *is included in the class of the fault-affected requests* in *run* if and only if

$$H_{run} = (e_0;\ x;\ e_u = send(o_i^r, o_j, rq_{l,p}^{o_j}, \text{false});\ y;\ e_v = resp(o_i^r, o_j^{r2}, rp_{l,p}^{o_j}, nf);\ w)$$

where x, y finite sequences of events with $e_u \notin x$, e_v is the first *resp* event with $r_2 \in \{ar_m \mid 1 \leq m \leq \#(o_j^r)\} \cup \{prim\}$, *nf* being either true or false and w is an infinite sequence of events such that either:

i. $S_v \models F\_AFFECTED(rq_{l,p}^{o_j}, o_j)$ or

ii. $S_v \models (\exists\, op_s^{o_i} \in MsgSeq(op_l^{o_j})$, $1 \leq s \leq \#(\text{methods of } o_i)$, $1 \leq i \leq n$, $i \neq j$ and $F\_AFFECTED(rq_{s,p}^{o_i}, o_t)$ for some $1 \leq t \leq n$, $t \neq j)$

iii. $\exists$ *resp*(o_t^{r1} , o_j^{r2} , $rp_{s,q}^{o_j}$, false)$\in$ *y* for some o_t^{r1} , $1 \leq s \leq$ #(methods of o_j) such that

$$S_v \models F\_AFFECTED(rq_{s,q}^{o_j} , o_u), \text{ for some } 1 \leq u \leq n$$

Events *resp*(o_i^{r} , o_j^{r3} , $rp_{l,q}^{o_j}$, *nf*)$\in$ *w* with $r_3 \neq r_2$ do not change the classification of $rq_{l,q}^{o_j}$.

The first condition reflects the case whereby $rq_{l,p}^{o_j}$ is affected by a fault at some o_j^{r} . The second condition reflects the case whereby one of the nested invocations generated by $rq_{l,p}^{o_j}$ is affected by a fault at o_t , for some $1 \leq t \leq n$, $t \neq j$. Finally, the third condition reflects the case whereby $rq_{l,p}^{o_j}$ is queued behind a $rq_{s,q}^{o_j}$ that is eventually affected by a fault at o_u , for some $1 \leq u \leq n$. All other $rq_{l,p}^{o_j}$ ($p \in \aleph$) in *run, are included in the class of the fault-unaffected requests*.

A service request is affected by the occurred faults, if at least one of the generated requests becomes fault-affected or if it is queued behind another service request that becomes fault-affected. We propose evaluation of fault-tolerance effectiveness and fault-tolerance performance based on the mean response times for the fault-affected and the fault-unaffected service requests. In our hybrid reliability and system's traffic simulation it is also possible to produce typical system reliability and service availability estimates. However, we believe that traffic-based measures that are defined separately for the fault-affected and the fault-unaffected service requests are more powerful in capturing the essence of the most influential fault-tolerance performance and effectiveness tradeoffs.

As a design mean, the proposed evaluation approach can be exploited in the following two ways:

- To determine the minimum fault-affected service times (*optimum effectiveness*) that a composite replication scheme can yield, for any possible combination of values for the considered replication parameters. In candidate schemes that are composed of possibly different replication policies, *their optimum effectiveness configuration is the only mean that makes feasible the comparison between them*. We prefer the selection of the composite replication scheme that fulfills the set QoS design goal at the lowest cost (best fault-tolerance performance).

- To determine appropriate values for the considered replication parameters, with respect to the set QoS design goals (not the optimum effectiveness ones). This is done by an appropriate *trade-off analysis*, where, for each potential change against the considered base replication scheme, we trade the potential improvements in the fault-affected service requests, against the overhead imposed to the fault-unaffected ones. For a composite replication scheme, such an analysis converges to the values combination that fulfills the set design goals at the lowest possible cost.

The proposed evaluation approach has been already exploited in [10], to found a systematic QoS design method that aims at the selection of appropriate checkpoint/state transfer intervals for the passively replicated objects. The optimum effectiveness configurations for the candidate composite replication schemes are determined by simulation metamodeling and optimization ([11]), in the frame of an appropriately selected uniform experimental design ([28]). Finally, the proposed trade-off decision-making procedure allows the selection of low-cost checkpoint/state transfer intervals, with respect to the set design goals.

In current paper, the formally specified evaluation approach and the developed prototype tool are used to give insight into another influential fault-tolerance performance and effectiveness tradeoff: the fact that excessively frequent request-retry timeouts for the constituent objects cause high overhead costs and do not improve fault-tolerance effectiveness.

5. A case system study

The results reported in this section unfold one of the tradeoff problems that are inherent in the design of a composite replication scheme: excessively frequent request-retry timeouts cause high overhead costs and do not improve fault-tolerance effectiveness. We aim to give insight in the selection of effective request-retry timeouts to mask (non-partitioning) network faults and different types of omission faults (from those described in section 3.2), where the client does not detect the fault and receives no reply. As we already noted, the description and experimentation with a systematic QoS design method (optimum effectiveness finding and trade-off analysis), which exploits the proposed evaluation approach, is treated elsewhere for a different trade-off problem.

The considered synthetic workload scenario includes an actively replicated object and allows for a range of combinations of omission faults and request-retry policies. The system model is comprised of four (4) stateless service objects (`obj0`, `obj5`, `obj6`, `obj7`) that are instances of the class `SrvRequestAccepting` and implement the provided service by invoking methods in four (4) different state owning objects (`obj1`, `obj2`, `obj3`, `obj4`) as shown in Figure 3. Received type-1 and type-2 service requests are assigned to the available service objects (`obj0`, `obj5`, `obj6`, `obj7`) on a round-robin basis (Table 2).

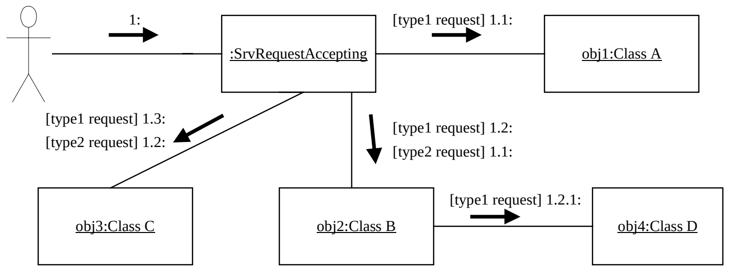


Figure 3 Objects' collaboration diagram

Table 2 System's computational setting

service objects:	objX:SrvRequestAccepting (X=0, 5, 6, 7)	no state
backend objects:	obj1, obj2, obj3, obj4	own state
multithreading:	thread-per-object	
service requests assignment (load balancing):	per-request load balancing: requests are assigned to objX (X=0, 5, 6, 7) on a round robin basis	

5.1 The simulated composite replication scheme and the system's traffic and object fault models

The four (4) stateless service objects (`obj0`, `obj5`, `obj6`, `obj7`), as well as `obj2`, `obj3` and `obj4` are replicated according to the warm passive replication policy (with a single backup) shown in Figure 2. `Obj1` is replicated according to the active replication policy (with two replicas) shown in Figure 1.

Table 3 summarizes the used system’s traffic and resource consumption parameters. Resource consumption depends on the speed and the load of the hosts, where the object servers are placed. Also, since we did not want to burden our model with extra parameters that are not related to the fault-tolerance performance and effectiveness tradeoffs, we assumed that the available network bandwidth is large enough, so that network latency variations as a consequence of bandwidth contention are not significant. Log-induced replayed requests do not cause re-execution of re-invoked requests, but result in a retransmission of the already computed responses. Finally, resource consumption for the checkpoints and the state transfers depends on the object state sizes and the computational capacity of the underlying hosts (state transfer speeds).

Table 3 System’s traffic and resource consumption parameters

system’s traffic parameters (exponential with means)										
type 1 request arrivals (sec)	2.5									
type 2 request arrivals (sec)	2.5									
object replicas: resource consumption parameters	**rep10 obj1**	**rep11 obj1**	**rep20 obj2**	**rep21 obj2**	**rep30 obj3**	**rep31 obj3**	**rep40 obj4**	**rep41 obj4**	**repX0 objX**	**repX1 objX**
type 1 requests service times (exponential with means)	0.52	0.52	0.25 (*)	0.25 (*)	0.7	0.7	0.32	0.32	0.05 (*)	0.05 (*)
type 2 requests service times (exponential with means)	-	-	0.28	0.28	0.7	0.7	-	-	0.05 (*)	0.05 (*)
log-replayed re-invoked requests (exponential with means)	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	-	-
object state sizes (KB)	0.9		0.7		0.5		0.6		-	-
state transfer speed -sec/KB (exponential with means)	0.8	0.8	0.6	0.6	0.6	0.6	0.8	0.8	-	-

(*) resource consumption prior to nested requests invocation and following the reception of the last reply

Table 4 summarizes the simulated replication scheme and the considered value combinations for its replication parameters. We give insight into: (i) the case of a request-with-no-retry policy that is not possible to mask (non-partitioning) network faults and omission faults and (ii) the considered three request-retry scenarios, which are accompanied by the related overhead costs for re-invocation of requests that are possibly lost.

Table 4 The simulated composite replication scheme

composite replication scheme	obj0	obj1:classA	obj2:classB	obj3:classC	obj4:classD	obj5	obj6	obj7
replication:	passive (fig. 2)	active (fig. 1)	passive (fig. 2)	passive (fig. 2)	passive (fig. 2)	passive (fig. 2)	passive (fig. 2)	passive (fig. 2)
behavioral properties								
number of replicas:	2	2	2	2	2	2	2	2
checkpoint/state transfer intervals (number of requests):	no state	-	60	30	90	no state	no state	no state
simulated request-retry scenarios (timeouts in sec)								
case I:	12.0	-	7.0	-	-	12.0	12.0	12.0
case II:	14.0	-	9.0	-	-	14.0	14.0	14.0
case III:	16.0	-	11.0	-	-	16.0	16.0	16.0

Table 5 summarizes the assumed object replicas allocation to the available object servers. Each server is placed on a separate host and the applied multithreading is (as specified in Table 2) the thread-per-object policy.

Table 5 Object replicas placement

object server 1	rep00 (obj0)	rep11 (obj1)	rep51 (obj5)
object server 2	rep01 (obj0)		rep50 (obj5)
object server 3	rep21 (obj2)	rep40 (obj4)	
object server 4	rep20 (obj2)	rep41 (obj4)	
object server 5	rep30 (obj3)		
object server 6	rep31 (obj3)		
object server 7	rep60 (obj6)	rep10 (obj1)	rep71 (obj7)
object server 8	rep61 (obj6)		rep70 (obj7)

Table 6 The simulated object fault models

fault rarity:	21600 sec									
object replicas:	repX0 objX	repX1 objX	rep10 obj1	rep11 obj1	rep20 obj2	rep21 obj2	rep30 obj3	rep31 obj3	rep40 obj4	rep41 obj4
fault interarrival times (exponential)	2*r	2*r	2*r	2*r	r	r	r	r	2*r	2*r
replicas restart times (exponential)	23.0	23.0	23.0	23.0	23.0	23.0	23.0	23.0	23.0	23.0
omission faults (as specified in section 3.2):										
case A:	-		-		-		-		-	
case B:	-		-		(d)		(d)		(d)	
case C:	-		-		(a)		(a)		(a)	
fault monitoring intervals (sec):	2.0, 4.0, 6.0, 8.0, 10.0, 12.0									

Finally, Table 6 specifies the simulated object fault models. We propose the use of parametric object fault models, since faults are by definition rare events and it is always useful, if not necessary, to report the sensitivity of the obtained results, with respect to the assumed fault rarity (*r*). The table specifies fault interarrival times, object replicas restart times and three considered omission fault scenarios. The considered fault detection settings are summarized by the tested fault monitoring intervals, which are accompanied by the overhead costs mentioned in section 3.4.

5.2 Simulation results

Special emphasis has been given to the credibility of the produced means, by the use of appropriate output analysis procedures.

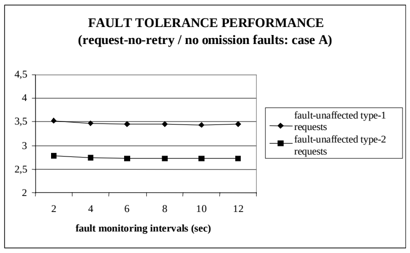
In case of passively replicated service objects we apply a single-run procedure ([18]) that exploits a representation of the required steady-state estimates in terms of quantities, which are based on the sample paths between two successive system entries into a selected set of states, say A. Such a target set of states for the mean response times in a service object, includes any state where the object's primary fails and the number of queued requests is 0. A-cycles are not independent and identically distributed and for this reason we use the batch means estimation. Successive A-cycle based quantities are grouped into non-overlapping batches and their means are treated as independent and identically distributed observations. The validity of this approximation increases with the batch size. The number of A-cycles and the batch size is determined dynamically, by the Law and Carson sequential control procedure ([12]), on the basis of the specified relative precision to be achieved.

For system configurations or system loads where it is not easy to identify a set of states, where system entries occur quite frequently (e.g. actively replicated service objects), we make use of the well-known independent replications approach.

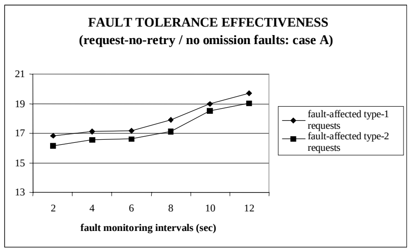
The results given in the following graphs were produced as 95% confidence intervals with half-width no more than 3% of the estimated value.

An appropriate composite replication scheme retains perspectives for trade-off decision-making between fault-tolerance performance and fault-tolerance effectiveness, with respect to the ever-changing QoS needs. In addition, fault-tolerance performance and effectiveness were found to depend on the system's load (requests arrival distributions). In the performed experiments, we assume the service request arrival distributions of Table 3.

Figure 4 presents the obtained mean fault-unaffected response times (performance) and mean fault-affected response times (effectiveness) for the two types of service requests and the composite replication scheme of Table 4, when there is no use of request-retry to mask potential (non-partitioning) network faults and omission faults. There is a notable improvement in fault-tolerance effectiveness (Figure 4b), when reducing the fault-monitoring interval from 12 sec to 6 sec, but we do not observe significant improvements in tighter fault detection settings. On the other hand, fault-tolerance performance (Figure 4a) seems to not be significantly affected by the overhead costs of the applied fault detection setting.



(a)



(b)

Figure 4 Fault-tolerance performance and fault-tolerance effectiveness for the request-no-retry replication scheme of Table 4

When applying a composite request-retry scheme to mask (non-partitioning) network faults, the overhead costs for re-invocation of requests that are possibly lost results in worse performance (Figures 5a, 5c, 5e) and fault-tolerance effectiveness (Figures 5b, 5d, 5f), when compared to Figure 4.

The request-retry scenario specified as case III in Table 4 (Figures 5a and 5b) allows exploiting fault monitoring intervals from 6 sec to 12 sec, without significant overhead costs. Tighter fault detection settings burden fault-tolerance performance (Figure 5a) with unacceptably high costs. When using more frequent request-retry timeouts (case II and case I in Table 4) for the constituent objects, we observe the same or higher overhead costs (Figures 5c and 5e respectively) and worse fault-tolerance effectiveness (Figures 5d and 5f respectively). Case I scenario is characterized by the use of excessively frequent timeouts that appear to result in a comparatively non-effective request-retry scheme.

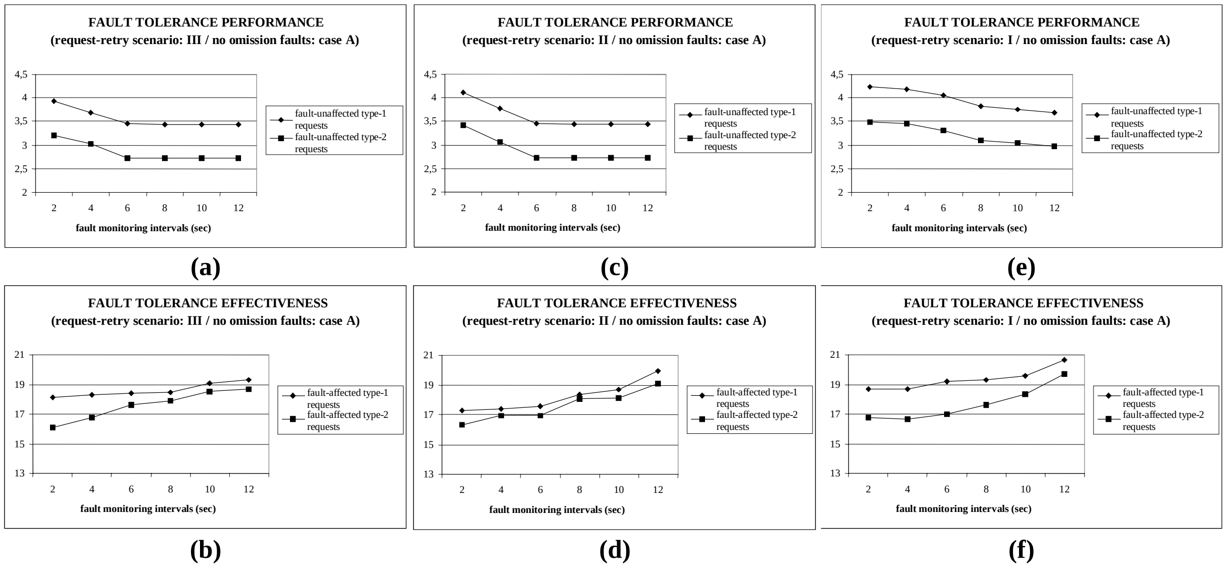


Figure 5 Fault tolerance performance and fault-tolerance effectiveness for composite replication schemes with different request-retry timeouts (Table 4)

Finally, we give insight into how performance and effectiveness are affected in different cases of simulated omission fault models (Figure 6).

Figures 6a and 6b refer to the case B scenario shown in Table 6: when a passively replicated object fails, the already queued requests are lost and the requests arriving while the object is down are also lost. We observe slightly improved performance (compared to Figure 5e) as a consequence of the empty queues found by the fault-unaffected requests arriving in the just recovered operational primary replicas.

Figures 6c and 6d refer to the case C scenario shown in Table 6: when a passively replicated object fails, no requests previously accepted in the queue are lost, but all requests arriving while the object is down are lost. Fault-tolerance performance (Figure 6c) is also improved, when compared to the no-loss case (scenario A of Table 6 and Figure 5e) and is slightly worse, when compared to the omission fault model of Figure 6a.

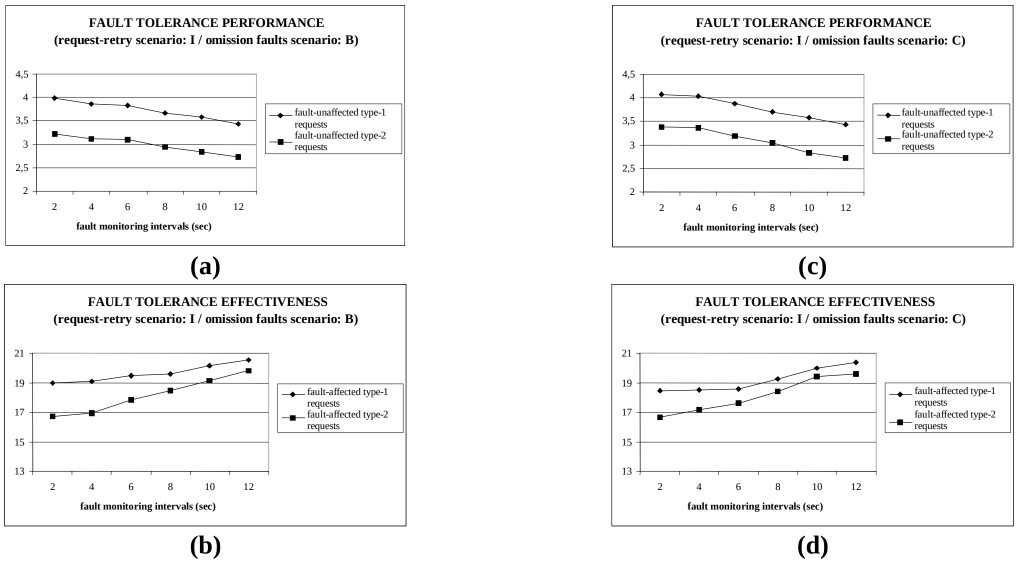


Figure 6 Fault tolerance performance and effectiveness under different omission fault scenarios (Table 6)

Different omission fault models reflect different possibilities of object fault handling for the used fault-tolerance infrastructure (see for example [1], [4], [8], [15], [17], [24]). We expect more significant differences in higher system load levels and this has to be taken into account, when designing composite replication and request-retry schemes that fulfill specific QoS goals.

6. Conclusion

We presented a quantitative evaluation approach for dependable server applications that possibly have to conform to agreed quality of service (QoS) guarantees for service response times. Compared to other reliability blocks based evaluation approaches we prefer hybrid reliability and system's traffic simulation and we focus on response time measures that are separately quantifying fault-tolerance performance and fault-tolerance effectiveness.

The proposed evaluation approach opens the following perspectives:

- to take into account complex interactions that are otherwise attributed to diverse design concerns (fault tolerance, load balancing and multithreading),
- to capture the essence of the most influential fault-tolerance trade-offs,
- to support a combined decision-making for replication parameters, such as checkpoint/state transfer intervals, request-retry timeouts and other,
- to provide estimates for candidate QoS goals that are often agreed between service providers and customers and
- to explore the perspective of a replication scheme for being adapted in ever-changing QoS needs.

The proposed evaluation approach can also be the cornerstone of systematic QoS design methods where (i) candidate replication schemes are compared on the basis of their optimum effectiveness configurations (the single criterion making feasible such a comparison) and (ii) it is possible to determine low-cost values for the replication parameters of the selected scheme, with respect to specific QoS design goals.

We believe that hybrid reliability and system's traffic simulation and the proposed evaluation approach constitute a valuable and generic tool possible to be exploited in the study of fault-tolerance performance and fault-tolerance effectiveness in many other contexts (e.g. coordinated checkpointing and message logging algorithms, transaction-based fault tolerance, component-based fault tolerance, as in [2] and [5] etc). Finally, the presented approach can also be the cornerstone of UML-based performance models ([16]) of dependable systems.

References

[1] T. Bennani, L. Blain, L. Courtes, J.-C. Fabre, M.-O. Killijian, E. Marsden, F. Taiani, Implementing simple replication protocols using CORBA portable interceptors and Java serialization, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 04), IEEE Computer Society Press, Florence, Italy, 2004, pp. 549-554

[2] I. Crnkovic, M. Larsson, Classification of quality attributes for predictability in component-based systems, Supplemental Volume of the 2004 International Conference on Dependable Systems and Networks, Florence, Italy, 2004, pp. 307-311

[3] O. Das, C. M. Woodside, Evaluating layered distributed software systems with fault-tolerant features, Performance Evaluation, 45, 1, 2001, pp. 57-76

[4] P. Felber, R. Guerraoui, A. Schiper, Replication of CORBA Objects, Distributed Systems, Lecture Notes in Computer Science 1752, Springer Verlag, 2000, pp. 254-276

[5] J. Fraga, F. Siqueira, F. Favarim, An adaptive fault-tolerant component model, Proceedings of the Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS’03), IEEE Computer Society, Capri Island, Italy, 2003, pp. 179-186

[6] S. Garg, Y. Huang, C. M. R. Kintala, K. S. Trivedi, S. Yajnik, Performance and reliability evaluation of passive replication schemes in application level fault tolerance, Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing, IEEE, Madison, Wisconsin, USA, 1999, pp. 322-329

[7] K. K. Goswami, R. K. Iyer, L. Young, DEPEND: A simulation-based environment for system level dependability analysis, IEEE Transactions on Computers, 46, 1, 1997, pp. 60-74

[8] R. Guerraoui, P. Eugster, P. Felber, B. Garbinato, K. Mazouni, Experiences with object group systems, Software: Practice & Experience, 30, 12, 2000, pp. 1375-1404

[9] E. J. Henley, H. Kumamoto, Reliability engineering and risk assessment, Prentice-Hall, 1981

[10] P. Katsaros, C. Lazos, Optimal object state transfer - recovery policies for fault tolerant distributed systems, Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 04), IEEE Computer Society, Florence, Italy, 2004, pp. 762-771

[11] P. Katsaros, E. Angelis, C. Lazos, Applied multiresponse metamodeling for queuing network simulation experiments: problems and perspectives, Proceedings of the 4th EUROSIM Congress on Modelling and Simulation, EUROSIM, Delfts, The Netherlands, 2001

[12] A. M. Law, J. C. Carson, A sequential procedure for determining the length of a steady state simulation, Operations Research, Vol. 27, 1979, pp. 1011-1025

[13] M. Lindermeier, Load management for distributed object-oriented environments, International Symposium on Distributed Objects and Applications (DOA'00), IEEE, 2000

[14] M. Litoiu, J. Rolia, G. Serazzi, Designing process replication and activation: a quantitative approach, IEEE Transactions on Software Engineering, vol. 26, no. 12, pp. 1168-1178, 2000

[15] V. Marangozova, D. Hagimont, An infrastructure for CORBA component replication, Proceedings of the 1st IFIP/ACM Working Conference on Component Deployment, Lecture Notes in Computer Science, Springer-Verlag, 2002, pp. 222-232

[16] M. Marzolla, Simulation-based performance modeling of UML software architectures, Dottorato di Ricerca in Informatica, II Ciclo Nuova Serie, Dipartimento di Informatica, Università Ca' Foscari di Venezia, 2003

[17] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, Strong replica consistency for fault-tolerant CORBA applications, Journal of Computer Systems Science and Engineering, CRL Publishing, 2002

[18] V. F. Nicola, P. Shahabuddin and M. Nakayama, Techniques for the fast simulation of models of highly dependable systems, IEEE Transactions on Reliability, 50, 3, 2001, pp. 246-264

[19] Object Management Group, Fault tolerant CORBA, OMG Technical Committee Document, 2001-09-29, September 2001

[20] Object Management Group, Object Management Architecture Guide, revision 3.0, OMG Technical Committee Document ab/97-05-05, June 1995

[21] Object Management Group, The Common Object Request Broker: Architecture and Specification, revision 2.3.1, OMG Technical Committee Document formal/99-10-07, October 1999

[22] H. S. Paul, A. Gupta, R. Badrinath, Performance comparison of checkpoint and recovery protocols, Concurrency and Computation: Practice and Experience, 15, 2003, pp. 1363-1386

[23] B. Ramamurthy, S. J. Upadhyaya, R. K. Iyer, An object-oriented test-bed for the evaluation of checkpointing and recovery systems, Proceedings of the 27th International Symposium on Fault-Tolerant Computing, IEEE, Seattle, WA, USA, 1997, pp. 194-203

[24] Y. Ren, D. E. Bakken, T. Courtney, M. Cukier, D. A. Karr, P. Rubel, C. Sabnis, W. H. Sanders, R. E. Schantz, M. Seri, AQuA: An Adaptive Architecture that Provides Dependable Distributed Objects, IEEE Transactions on Computers, Vol. 52, No. 1, 2003, pp. 31-50

[25] R. D. Schlichting, F. B. Schneider, Fail-Stop processors: An approach to designing fault-tolerant computing systems, ACM Transactions on Computer Systems, 1, 3, 1983

[26] D. C. Schmidt, Evaluating architectures for multithreaded object request brokers, Communications of the ACM, vol. 41, no. 10, pp. 54-60, 1998

[27] T. Schnekenburger, Load balancing in CORBA: A survey of concepts, patterns and techniques, The Journal of Supercomputing, 15, 141-161, Kluwer Academic, 2000

[28] Uniform Design web pages, http://www.math.hkbu.edu.hk/UniformDesign/, 2000