Performance tuning policies for application level fault tolerance in distributed object systems

Theodoros Soldatos\textsuperscript{a,*} and Nantia Iakovidou\textsuperscript{b}

\textsuperscript{a}EMBL Heidelberg, Meyerhofstrasse 1, D-69117 Heidelberg, Germany
\textsuperscript{b}Department of Informatics, Aristotle University of Thessaloniki, 54124 Thessaloniki, Greece

Abstract. In distributed object systems, application level fault tolerance is often attained by appropriate object replication policies. These policies aim at increasing the exhibited service availability by masking potential faults that do not recur after recovery. Existing middleware support infrastructures allow customizing object replication properties. However, since fault tolerance has a significant impact in the perceived service performance, there is a need for a suitable quantitative design technique, which allows comparing different replication policies by trading off the caused overhead cost against the achieved fault-tolerance effectiveness. We are also interested in taking into account different concerns in a combined manner (e.g. fault tolerance combined with load balancing and multithreading). This paper presents experimental evidence for the most important performance tradeoffs revealed in a simulation-based study. We considered different cases of object request loss behavior for the faulty objects, as well as, a number of request-retry strategies. The experiments took place in two different application workload levels for varied fault detection settings. We provide results for the combined effects of the studied replication policies with two specific load-balancing strategies. The presented results constitute a valuable experience report for performance tuning object replication policies for application level fault tolerance.

Keywords: Fault-tolerance, software replication, dependable systems

1. Introduction

Software replication is a well-known technique used to increase the availability of systems and services. Object replication has been standardized in a plain specification (OMG FT-CORBA in [7]) that prescribes how to apply operations in multiple replicas of an object in a transparent and consistent manner. OMG FT-CORBA also refers to a range of object replication properties that can be easily customized in most existing middleware support infrastructures [1,8,9]. This allows development of appropriate replication policies for application level fault tolerance.

Recently, in [3,10] the authors introduced a quantitative design technique, which allows comparing checkpoint-based replication policies by trading off the caused overhead cost against the achieved fault-tolerance effectiveness. It was found that fault tolerance has a significant impact in the perceived service performance and this impact depends on the applied workload intensity and the (possibly different) replication policies for the interacting objects. Fault tolerance overhead and effectiveness are affected by

\*Corresponding author. E-mail: soldatos@embl.de.
the interactions between the application objects, as well as, by the combined effects of other design factors such as load balancing and multithreading. However, there is little experimental evidence regarding the performance tradeoffs that arise when applying replication-based fault tolerance in distributed object systems.

Object replication lies 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. Object replication is used in minimizing loss of computation in the presence of non-recurrent faults.

Failures that do not recur after recovery may be caused by insufficient memory, media failures, power outages, non-partitioning network failures and the non-determinism introduced by distributed timers and the use of multithreading. We consider three distinct possibilities when a server object fails and two types of behavior for the period that a server object is unavailable. The three possibilities regarding the occurrence of an object fault are:

- all queued object requests are lost,
- only the request in service (if any) is lost and
- no request is lost.

The two types of behavior for the period that a server object is unavailable are:

- the server object continues to accept arriving object requests, while it is not available for service and
- the server object does not accept incoming object requests, while it is not available.

According to the OMG FT-CORBA specification the replication policy assigned to an object can be either active, warm passive or cold passive replication and each one of the aforementioned policies is customized by a set of behavioral properties (number of replicas, checkpoint/state transfer interval, request-retry timeout etc.).

In this paper, we attempt to give insight into the most influential performance tradeoff problems that appear in composite object replication schemes. The experiments were performed by a combined reliability and system’s traffic simulator and more precisely by the one described in [5]. This simulator implements the performance evaluation approach employed in the quantitative design technique of [3].

Investigation of the found performance tradeoffs took place with regard to a synthetic workload scenario, where we had the chance to apply fault-tolerance schemes with different fault occurrence behaviors, request-retry timeout intervals, requests queue buffer sizes etc. The experiments were performed in two different workload levels for varied fault detection settings.
The rest of the paper is organized as follows. Section 2 outlines the functionality of the used simulator. Section 3 describes the considered case system model and its parameters. Section 4 presents and comments the obtained results. Finally, the paper concludes with a summary of the findings, which we believe provide valuable guidance for performance tuning object replication policies for application level fault tolerance.

2. The used simulator

The used hybrid reliability and system’s traffic simulator allowed us to take into account the interaction effects regarding:

- the simultaneous resource possession, caused by the synchronous and often nested object invocations,
- the hardware resource contention, as a result of the replicas distribution,
- the extra load and the queued requests blocking costs caused by the recurrent checkpointing/state transfer activities for the passively replicated objects (if any),
- the extra load caused by a replica restart and replay of the logged requests (if any),
- the overhead assumed for re-invocation of requests that are possibly lost because of a (non-partitioning) network fault or because of a recipient omission fault and
- the overhead assumed for periodically polling if an object is faulty or not.

In active replication, all object replicas execute incoming invocations independently, but in the same order. This ordering ensures that the states of the replicas are consistent at the end of each individual operation.

For a passively replicated object there is a single designated replica, known as the primary that performs all the invoked operations. The state of the primary and the sequence of the invoked methods are recorded.
in a log, according to the specified checkpoint properties. All the other object replicas are called backups and they do not issue or receive invocations and responses, while the primary replica is operational. Their sole purpose is to provide a pool of replicas from which a new primary can be chosen, if the current primary fails. In cold passive replication (CPR), the backup replicas are not even loaded into memory (activated) and thus they do not come into existence until the primary replica fails. In warm passive replication (WPR), the backup replicas are already created and initialized and the state of the primary is retrieved and transferred to all of the backup replicas in a frequency specified as an object replication property.

The modeled fault detection mechanism assumes the existence of a transparent and fault tolerant fault monitoring service. Each object is periodically checked according to a specified time interval. Fault monitoring has been found [11] to incur an approximate 5% increment, in the processor utilization, for about 500 milliseconds. This overhead has been taken into account.

The simulator supports request re-invocation in the following contexts:

- As a means utilized by a fault tolerance infrastructure or a client application to mask recipient faults and (non-partitioning) network faults, where the client-side does not detect the problem and receives no reply. The simulator allows the use of alternative request-retry timeouts and evaluates the accompanied overhead costs.
- In the course of a log-based recovery of a stateful object, when replaying all nested object requests.

Both cases of request re-invocations conform to the at-most-once invocation semantics of the CORBA object model, which means that each request is executed at most once. However, in the second case the recipient object replies with the result of the already invoked method, which incurs a certain processing cost for the server object. Duplicate message requests as a consequence of the employed replication scheme are detected and suppressed [1,2,6].

Fig. 3. Fault affected response times (sec) – Requests arrival rates 2.5 & round-robin load balancing.
3. The case system model

The used case system model is comprised of four (4) stateless service objects (instances of the class SrvRequestAccepting) and four (4) state owning objects (obj1, obj2, obj3, obj4) interacting as shown in Fig. 1. Received class-1 and class-2 requests are assigned to the available service objects based on the applied load balancing strategy (round-robin or random probabilistic with equal probabilities).

The four (4) stateless service objects as well as obj2, obj3 and obj4 are replicated according to the warm passive replication policy – with a single backup. Obj1 is actively replicated by utilizing two (2) object replicas.

The resulted composite fault tolerance scheme allows testing any combination of
– the scenarios mentioned in section 1 regarding the occurrence of an object fault and
– the two types of behavior also mentioned in section 1, for the period that a server object is unavailable, with a request-no-retry policy or a range of request-retry timeouts for the constituent objects.

Table 1 describes the initial replicas function and the placement of the sixteen object instances to the available process nodes. Collocated object replicas are processed in a processor sharing discipline and each of them is placed on a separate object server. Table 2 specifies the system model parameters, the considered workload intensities and the applied load balancing strategies.

Table 3 summarizes the simulated combinations of fault-occurrence scenarios. The applied combination characterizes the fault-tolerance infrastructure used for the provision of the required level of dependability. The conducted experiments included the following three cases:
– **Loss scenario 1**: No request is lost on occurrence of an object fault and the object continues to accept incoming requests while it is not available.
– **Loss scenario 2**: All queued requests are lost on occurrence of an object fault and the object does not accept incoming object requests while it is not available. This is the default loss scenario for the object replicas of an actively replicated object.
Table 1
Placement of objects’ replicas

<table>
<thead>
<tr>
<th>Process node</th>
<th>Object 0 (backup)</th>
<th>Object 1 (1st replica)</th>
<th>Object 5 (primary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Object instances placement:
- process node 1: object 0 (backup) object 1 (1st replica) object 5 (primary)
- process node 2: object 0 (primary) object 5 (backup)
- process node 3: object 2 (backup) object 4 (backup)
- process node 4: object 2 (primary) object 4 (primary)
- process node 5: object 3 (backup)
- process node 6: object 3 (primary)
- process node 7: object 6 (primary) object 1 (2nd replica) object 7 (backup)
- process node 8: object 6 (backup) object 7 (primary)

Table 2
Model parameters

<table>
<thead>
<tr>
<th>Object</th>
<th>State size (in KB)</th>
<th>No of invocations between checkpoints/ state transfers</th>
<th>Fault process speed (exp. with rates)</th>
<th>State transfer speed (sec/KB)</th>
<th>Class1 service (exp. with rates)</th>
<th>Class2 service (exp. with rates)</th>
<th>Reinvoked requests (exp. with rates)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No state</td>
<td>2r</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>1</td>
<td>0.9</td>
<td>AR</td>
<td>2r</td>
<td>0.8</td>
<td>0.52</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>0.7</td>
<td>60</td>
<td>r</td>
<td>0.6</td>
<td>0.25</td>
<td>0.25</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.5</td>
<td>30</td>
<td>r</td>
<td>0.6</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
<tr>
<td>4</td>
<td>0.6</td>
<td>90</td>
<td>2r</td>
<td>0.8</td>
<td>0.32</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>No state</td>
<td>2r</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>6</td>
<td>No state</td>
<td>2r</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
<tr>
<td>7</td>
<td>No state</td>
<td>2r</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Object replicas restart times are exponential with rate 23.0 seconds.
Class 1 and class 2 requests arrivals are exponentials with rates: (i) 2.5 seconds and (ii) 2.3 seconds for both of them.
Request assignment to the available service objects based on (i) round robin or (ii) random probabilistic with equal probabilities.
r = fault rarity = 21600 seconds

– Loss scenario 3: No queued requests are lost on occurrence of an object fault, but the object does not accept incoming object requests while it is not available.

4. Performance tradeoffs and obtained results

Application level fault tolerance is ruled by a set of tradeoffs:

– Excessive checkpointing and/or frequent state synchronizations result in performance degradation, since in the course of a checkpoint placement or a state transfer, incoming requests cannot be processed before its end. On the other hand, deficient checkpointing and infrequent state synchronizations incur expensive recovery.

– Frequent request-retry timeouts and tight fault-monitoring intervals result in increased overhead costs, as opposed to infrequent ones, which may cause high requests response times.

In addition to the aforementioned tradeoffs, fault tolerance performance also depends on the structural dependencies imposed by the objects’ invocation flows. The employed simulator produces appropriately designed performance metrics [3], which allow trading the gains of a potential replication properties change, against the imposed overhead.

This is achieved by providing separate response time statistics for

– the service requests that are not affected by the occurred faults (fault unaffected) from
Table 3
Simulated fault tolerance schemes

<table>
<thead>
<tr>
<th>EXPERIMENTS</th>
<th>objects 0, 5, 6, 7</th>
<th>object 1</th>
<th>object 2</th>
<th>object 3</th>
<th>object 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>loss scenario</td>
<td>I</td>
<td>1</td>
<td>(2)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>request-retry timeouts (in sec)</td>
<td>II</td>
<td>12</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>loss scenario</td>
<td>III</td>
<td>12.0</td>
<td>7.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>request-retry timeouts (in sec)</td>
<td>IV</td>
<td>12</td>
<td>7.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>loss scenario</td>
<td>V</td>
<td>14.0</td>
<td>9.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>request-retry timeouts (in sec)</td>
<td>VI</td>
<td>16.0</td>
<td>11.0</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Fault monitoring intervals (in sec):</td>
<td>2.0, 4.0, 6.0, 8.0, 10.0, 12.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 5. Fault affected response times (sec) – Requests arrival rates 2.3 & round-robin load balancing.

– the service requests that are affected by the occurred faults (fault-affected).

Fault-unaffected requests constitute the vast majority of the dispatched service requests, because faults are by definition rare events. However, response time guarantees quite often include the response times of the fault-affected requests that in fact provide an adequate measure of the achieved fault tolerance effectiveness.

The checkpoints placement tradeoff problem has been studied in [3]. In present paper, we provide revealing simulation results regarding the effectiveness and the performance of the composite request-retry policies and loss scenario combinations of Table 3.
Figure 2 summarizes the simulation results for the fault unaffected response times, for request arrival rates 2.5 and round-robin load balancing. Figure 4 shows the results obtained for request arrival rates 2.3.

The fault tolerance scheme with the request-no-retry policy (SYSTEM I) does not exhibit significant overhead differences for the tested fault monitor intervals. However, we cannot use the request-no-retry policy for masking e.g. requests losses or queue buffer overflows in the server objects, as well as for masking non-partitioning network faults.

The fault tolerance scheme of SYSTEM IV is characterized by an extra overhead cost (as expected) and a steep increase of it, when using more effective fault-monitoring settings. This higher overhead is due to the continuously occurring timeouts in the senders, while the sent requests are queued in the recipients.

When the already queued requests are lost at the time a server object fails and the incoming requests while the object is down are also lost (SYSTEM II), the overhead cost for the fault-unaffected requests is significantly lower. This comes as a consequence of the empty queues found by the (fault-unaffected) requests arriving in the just recovered operational primaries of the passively replicated objects.

When no queued requests are lost at the time a server object fails, but only incoming ones are lost (SYSTEM III), we obtain improved overhead costs compared to the case of SYSTEM IV and worse than in the case of SYSTEM II.

When the applied loss scenario is combined with less frequent request-retry timeouts (SYSTEM V & SYSTEM VI), we observe expanded possibilities to exploit more effective fault detection settings, without additional overhead (fault-monitoring intervals from 12.0 to 6.0 sec). In addition, as the request-retry timeouts become less frequent, the performance tends to be the same as in SYSTEM I, where there is no request-retry policy. However, fault tolerance performance does not improve when increasing the used
request-retry timeouts more than an appropriate threshold. The selection of a composite request-retry policy with suitable timeout settings should be the subject of a systematic quantitative design method like the ones described in [3,4].

A comparison with the results shown in Fig. 4 confirms that fault tolerance overhead depends on the applied workload intensity. Requests arrival rates 2.3 result in performance degradation for all simulated system configurations. This trend is more intense in the case of SYSTEM I, due to the absence of request-retry timeouts. However, both workload intensities seem to not be close to the system thrashing point, where we expect higher differences from the ones shown.

Fault tolerance effectiveness is quantified by the means of the fault-affected requests shown in Fig. 3 (requests arrival rates 2.5) and 5 (requests arrival rates 2.3). All depicted means were obtained along with 95% confidence intervals with half width no more than 3% of the estimated value. We thus ensured statistically significant samples for the fault-affected requests (faults are by definition rare events).

Regarding the effectiveness of the tested configurations we draw the following conclusions:

– Excessively frequent request-retry timeouts do not improve fault-tolerance effectiveness due to the incurred overhead costs. This is shown when comparing the graphs of systems I, V and VI to the graphs of systems II, III, IV.

– The applied loss scenario has a slight impact on the resulted fault-tolerance effectiveness and this is obvious, when comparing the graphs of systems II and IV for the tightest fault monitoring intervals (from 2.0 and 4.0 sec).

The obtained fault affected means for requests arrival rates 2.3 (Fig. 5) are worse due to the additional fault tolerance overhead. However, the applied workload intensity does not seem to have significant impact on the tradeoffs imposed by the employed combination of loss scenarios, request-retry timeouts and fault-monitoring intervals.

The results obtained for random probabilistic load balancing (with equal probabilities for the used service objects) exhibit the impact of the applied request assignment policy to the incurred fault tolerance overhead. We observe (Fig. 6) that apart from the shown performance degradation, when compared to the round robin results, the later exhibit a more significant improvement for the fault affected response times (fault tolerance effectiveness) as the fault monitoring interval decreases.

5. Conclusion

In this paper, we addressed the most important performance tradeoff problems that have to be taken into account in the design of object replication based fault tolerance. In general, the perceived performance and fault-tolerance effectiveness depend on the complex structural dependencies imposed by the objects’ invocation flows.

However, the findings of the performed simulation experiments suggest that they also depend on the applied workload intensities, on the effects of design factors other than fault tolerance, like for example load balancing, as well as on the overhead and effectiveness tradeoffs incurred by the applied composite replication policy. We explored the most influential performance tradeoffs for different cases of object request loss behavior for the faulty objects and for a range of alternative request-retry strategies.

Although we did aim to provide a systematic quantitative design technique, the presented results constitute a valuable guide for performance tuning object replication policies for application level fault tolerance.
References


