

Performance Analysis of Java Group Toolkits: A Case Study*

Roberto Baldoni, Stefano Cimmino, Carlo Marchetti and Alessandro Termini

Dipartimento di Informatica e Sistemistica,
Università di Roma “La Sapienza”
Via Salaria 113, 00198, Roma, Italy
email: {baldoni, cimmino, marchet, termini}@dis.uniroma1.it

Abstract. *In this paper we present a performance evaluation of three Java group toolkits (JavaGroups, Spread and Appia) with respect to a specific application scenario: software replication using a three-tier architecture. We also compare performances of these Java group toolkits with Maestro/Ensemble which has been developed in C++. Results show that performances of Java group toolkits are becoming comparable with the ones of Maestro/Ensemble, once selected a well tailored protocol stack for a given application.*

Keywords: *Group Communications, Java, Software replication, Performance.*

1 Introduction

There is an emerging demand for Java-oriented reliable software for distributed applications due to Java platform independence, the dynamic plug-in of Java components (e.g. downloading Java applets) and the simplicity of application programming just to name some of the most important Java features. One of the well-known way to add reliability to a distributed application is through software replication. However there are only a few systems that implement replication in Java.

Software replication is a well known technique allowing to increase the availability of a service exploiting specialized software running on COTS (Commercial-Off-The-Shelf), cheap hardware. The basic idea underlying software replication is to replicate the server of a given service on different hosts connected by a communication network so that the service’s clients can connect to different server *replicas* to increase the probability of getting replies to their requests. When dealing with a replicated service, it arises the problem of guaranteeing *consistency* among the local states of the replicas despite crashes. *Active* [1] and *passive* [2] replication [3] are well-known approaches to increase the availability of a stateful service. These approaches employ group communication primitives and services such as *total order multicast*, *view synchronous multicast*, *group membership*, etc. Implementations of such primitives are provided by *group communication toolkits* e.g., ISIS [4], TOTEM [5], Maestro/Ensemble [6]. Previous group toolkits have been implemented in C and/or C++. Only recently Java group toolkits have emerged e.g. JavaGroups [7], Spread [8, 9] and Appia [10]. However, to the best of our knowledge, there does not exist a performance comparison among these toolkits, and the evaluation of Java implementations is a critical point as distinct Java designs of the same application can lead to huge differences in performance due to the interaction

* This work has been partially supported by a grant from EU IST Project “EU-PUBLIC.COM” (#IST-2001-35217)

between the virtual machine and the bare operating system and the network. Furthermore, a performance comparison among such group toolkits cannot prescind from a specific application, otherwise the comparison risks to be useless.

In this paper we present a performance evaluation of Java group toolkits in the context of a specific architecture for software replication, namely the three-tier (3T) software replication, based on group toolkit. More specifically, in a three-tier architecture for software replication clients (the client-tier) interact with a middle tier (the middle-tier) that forwards client requests to replicas (the end-tier) maintaining consistency. To achieve this, the middle-tier embeds two basic components, namely the *sequencer* and the *active replication handler*. The first component assigns, in a persistent way, consecutive and unique sequence numbers to client requests. This is based on the classical "one-shot" total order provided by group toolkits¹, while the second masters the client/server interaction enforcing atomicity on the end-tier.

Therefore to evaluate Java group toolkits we realized an implementation of the sequencer component, one for each group toolkit, and measured their performance in terms of client latency and sequencer component latency. We show how the performance of a given Java group toolkit is heavily influenced by the protocol implementing the total-order primitive. To evaluate the gap between a C++ group toolkit and Java ones, we also show the performance results of the same 3T architecture whose sequencer is based on Maestro/Ensemble. This performance comparison points out that, once selected the "one-shot" total order protocol well tailored for the underlying application, the gap between a Java group toolkit and Maestro/Ensemble is acceptable.

The rest of the paper is structured as follows: Section 2 introduces main features of Java group communication toolkits. Section 3 presents the application scenario, namely software replication based on a three-tier architecture, where group toolkits will be evaluated and finally Section 4 shows the performance results.

2 Group Communication Toolkits

Group Communication Toolkits (GCTs) have proven to be an effective tool for building reliable software in partially synchronous distributed systems². They provide a rich set of services and primitives (e.g., total order, group membership etc) which help developers in adding specific *properties* to their application. As an example, they can be used to build highly available systems through software replication. The advent of Java has raised the need to provide Java GCTs, which exploit the advantages of this language. However, advantages like portability, ease of programming and adaptivity often negatively impact performance; in particular, for a Java application, the way the virtual machine interacts with the underlying operative system and network can greatly influence performance. In order to compare different toolkits from a performance point of view, we focus on differences in their architecture and implementations. To this end, as a first abstraction level, we identify two basic building blocks in the architecture of a GCT, namely (i) the API and (ii) the core system, as shown in Figure 1(a). The first one is the set of interfaces used by the application to use the GCT services; the second is the part that actually implements such services. With respect to these building blocks, GCTs can be classified according to (i) the programming language used and to (ii) the

¹ "One-shot" means that there is no relation between any two consecutive runs of a total order protocol.

² A partially synchronous distributed system alternates stable and unstable intervals. During a stable interval there is a bound (known or unknown) on (i) communication transfer delays and (ii) on the time a process takes to execute a step. During unstable intervals the system becomes purely asynchronous. If the duration of stable intervals tends to zero, the group toolkit does not longer guarantee its services. Conditions verifying partial synchrony can be easily ensured on a LAN.

deployment of the two blocks. In fact the API and the core system can be implemented in different languages; moreover, they can either reside in the same process, thus sharing the same address space, or they can run in two different processes on the same host, communicating through IPC, or they can even be deployed on different hosts, communicating through the underlying network. Moreover as we are interested in Java toolkits, it is important to consider where the Java Virtual Machine (JVM) is located with respect to a given deployment. In the case in which only the API is written in Java, the JVM is collocated under this block, while the core is free to exchange messages with the network using native system calls. On the contrary, if even the core is written in Java, the communication with the network passes through the JVM, which is collocated under the core. Let us remark that for every invocation of a service done by an application through API, the core system often needs to exchange several messages. Therefore, having the core written in Java can yield some performance penalties. Concerning the

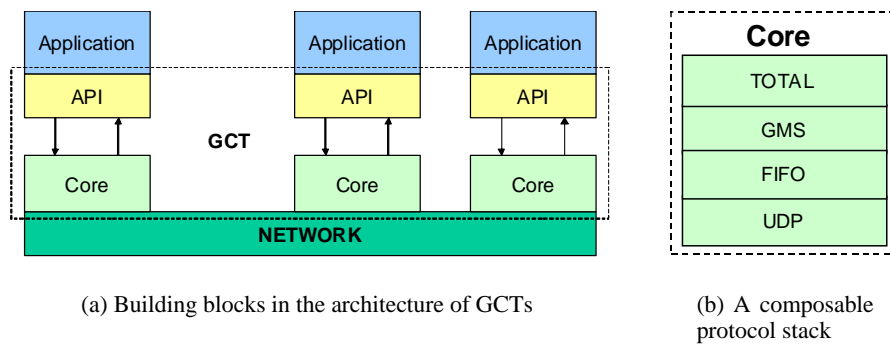


Fig. 1. Architecture of group communication toolkits

collocation of the two blocks, we should note that if they reside in different processes or in different hosts, there are two additional hops in the communication between any two processes of a group, thus yielding an additional overhead that results in a performance degradation. Another abstraction level regards the core system itself; in particular, we consider the protocols implemented within the core and the way in which they interact. From this point of view, GCTs can be classified into two distinct types: one is characterized by a *fixed* set of protocols, which interact always in the same way; the other is based on the concept of a *composable* protocol stack (see Figure 1(b)), where each specific functionality, e.g. total ordering of messages, is confined in a *micro-protocol*, which can be placed above another to form a stack. The developer is also free to add new semantics, by encapsulating them in a new micro-protocol, which can be used to compose another protocol stack. Moreover, this composition can be done dynamically, giving the possibility to build *adaptive systems*. As an example, an application sending totally ordered messages using the stack depicted in Figure 1(b), can decide, at some point in time, to add an encryption layer, defined by the developer, to obtain confidentiality. This behavior is not possible if the core system is built as a fixed predefined set of protocols, because it allows the developer to choose only among a static set of GCT services, whereas every additional functionality must be added at the application level, which is not easy and sometimes even impossible. In contrast, from a performance viewpoint, composable protocol stack can yield penalties due, for example, to arbitrary delays introduced by micro-protocols and to the growth of the message size due to layering.

We now give a brief description of the GCTs we used in the experiments, focusing on aspects that allow to classify them on the basis of the previous discussion.

Maestro/Ensemble [11]. Ensemble is a flexible and efficient toolkit written in the Objective CAML language. It is based on the concept of composable protocol stack, but also implements some optimizations, trying to overcome the performance degradation that results from a layered architecture. Maestro is an interface to Ensemble written in C++, and thus can be considered the API of the toolkit with respect to our framework defined above. Maestro starts Ensemble in a separate thread, therefore the API and the core reside in the same process, sharing the same address space.

Spread [8, 9]. The Spread toolkit is intended to be used in wide area networks, but it has also excellent performances if used in LANs. It is based on a client-daemon architecture, where the daemon represents the core system. The client connects to the daemon by means of a predefined API, and sends messages to other members of a group exploiting the daemon itself. Therefore the API and the core reside in different processes, which can be collocated on the same host, or in distinct ones. The daemon employs a fixed number of protocols, therefore, even if it gives a certain amount of flexibility in the services it offers, the developer is forced to add every additional functionality at the application level. The daemon is written in ANSI C, whereas the API is available in C++, Java and other languages.

Appia [10]. As Ensemble, the Appia toolkit is based on a protocol stack, but it is more flexible than Ensemble as it provides the possibility to extend not only the protocols that compose the stack, but also the events used for intra-stack communication. In Appia there isn't a clear separation between the API and the core system, because an application will make use of the toolkit by providing its own micro-protocol, and collocating it on the top of the stack. It follows that the application reside in the same process of the core system. The entire toolkit is written in Java.

JavaGroups [7]. JavaGroups is also entirely written in Java, with the aim to exploit as much as possible all the benefits of this language. It can be considered as an API at all, because it can be easily extended to be integrated with any group communication toolkit. As an example, it provides interfaces to Ensemble as well as its native core system. It is thus based on a protocol stack. In this configuration, the application and the core system run in the same process. An important remark on JavaGroups is that it is heavily based on *patterns*.

Table 1 summarizes previous discussion.

Toolkit	API language	Core language	Collocation of the JVM	Core design
Maestro/Ensemble	C++	OCaml	-	Composable
Spread	Java	ANSI C	Under the API	Fixed
Appia	Java	Java	Under the core	Composable
JavaGroups	Java	Java	Under the core	Composable

Table 1. Classification of some group communication toolkits

3 A Case Study: Software Replication

In this section we present the case study used for the group toolkits evaluation. We first briefly introduce software replication. Then we present the notion of three-tier replication, an overview of a protocol for three-tier *active* replication that has been used as a testbed of the performance evaluation of GCTs.

3.1 Software Replication

Software replication is a well-known technique to increase the availability of a service. The basic idea underlying software replication is to replicate the server of a given service on different hosts connected by a communication network so that the service's clients can connect to these different server *replicas* to increase the probability of receiving replies to their requests. The problem with the replication is to guarantee the *consistency* of the replicas; to this aim, it is firstly necessary that every replica produces the same result to the same client request. This is what the *linearizability* [12] consistency criterion formally states. Under a practical point of view, sufficient conditions to linearize the executions of a stateful replicated service are (i) *atomicity* and (ii) *ordering*. Atomicity means that either all or none replicas execute each state update; ordering means that replicas execute updates in the same order. Group toolkits help developers to enforce these conditions. As an example, a total order primitive ensures that every message is delivered to all or none the members of a group and that all members receive messages in the same order.

3.2 Three-tier Replication

The idea behind three-tier (3T) software replication is to free clients and replicas from participating to protocols that guarantee linearizability. This is achieved by embedding the replication logic (handling atomicity and ordering) within a software middle-tier *physically detached* from both clients and replicas (see Figure 2). In other words, the middle-tier encapsulates all the synchrony necessary to get linearizable executions of a stateful replicated service. The main advantage of the 3T architecture is, therefore, the possibility to deploy clients and servers in a pure asynchronous distributed system such as Internet. In this architecture, a client sends its request to the middle-tier, which

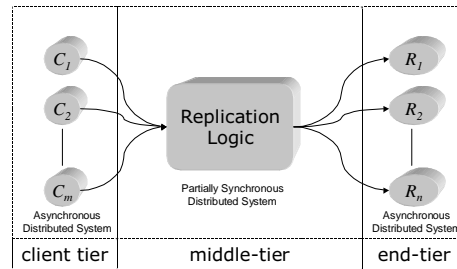


Fig. 2. 3T Architecture for Software Replication.

forwards it to replicas according to the replication logic implemented by the middle-tier. Then some replica processes the request and returns a result to the middle-tier that finally forwards the result to the client. The middle-tier has to be fault tolerant to ensure the termination of a client/server interaction in presence of failures. In particular, if a middle-tier entity that was carrying out a client/server interaction crashes, another middle-tier entity has to conclude the job in order to enforce end-tier consistency despite failures. Interested readers can find additional details on 3T replication in [13–15]. Figure 3 shows the components of the three-tier architecture for active replication (i.e., each replica executes the same set of client requests in the same order). In the remainder of this section we introduce a brief functional description of each component.

Retransmission/Redirection (RR). To cope with ARH replica failures and with the asynchrony of communication channels, each client process c_1, \dots, c_l embeds a

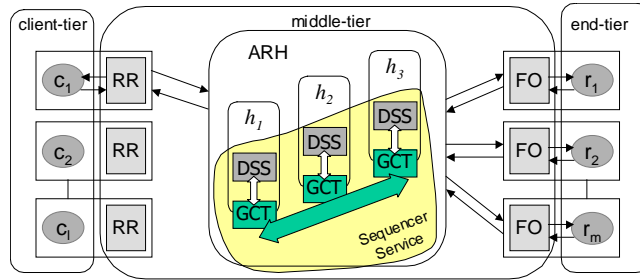


Fig. 3. A Three-tier Architecture for Active Replication

RR message handler. Clients invoke operations through RR that issues uniquely identified request messages to ARH. After the elapsing of a timeout set upon the request sending, RR retransmits the request, until a result is eventually received.

Active Replication Handler (ARH) component. ARH component is the core of the replication logic: by exploiting the *sequencer service*, it orders all incoming client requests and ensures that at least one copy of each ordered client request is eventually delivered at every available end-tier replica. Requests are sent to end-tier replicas along with the sequence numbers provided by the sequencer. replicas execute requests according to the sequence numbers. Once replicas return results, ARH returns the latter to clients.

Sequencer Service. The sequencer service is available to each ARH replica. In particular, each ARH replica has access to the distributed sequencer service (DSS) class which is a distributed and fault-tolerant implementation of the persistent sequencer service. This service returns an unique and consecutive sequence number for each *distinct* client request through the invocation of the *GetSeq()* method. As shown in Figure 3, each DSS component uses a GCT module as communication bus: in particular our implementation is based on the usage of a "one-shot" *total order multicast* primitive. DSS uses this primitive to assign a sequence number to a client request. In other words the DSS component manages a persistent state composed of pairs $\langle \text{client request}, \text{sequence numbers} \rangle$ that allows, for example, to retrieve, upon the failure of some component of the 3T architecture, a client request given a sequence number or viceversa [16].

Filtering and Ordering (FO). FO is a message handler placed in front of each end-tier replica (i) to ensure ordered execution of client requests according to the number assigned by DSS to each request, and (ii) to avoid repeated execution of the same client request (possibly sent by ARH).

Figure 4 illustrates a simple run of the protocol in which no process crashes. In this scenario, client c_1 invokes the method op_1 . Upon receiving req_1 , h_1 invokes the DSS component (*GetSeq()* method) to assign it a unique sequence number (1 in the example). This method embeds an invocation to the total order primitive of the group toolkit. Then h_1 sends a message containing the pair $\langle 1, op_1 \rangle$ to all end-tier replicas and starts waiting for the first result from an end-tier replica. When this reply arrives, h_1 sends, in its turn, a reply message back to c_1 . h_1 discards further replies to operation op_1 produced by end-tier replicas (for simplicity these messages are not shown in Figure 4). Concurrently, h_2 serves req_2 sent by c_2 . A prototype of the 3T architecture for software replication, namely the Interoperable Replication Logic [17], based on a CORBA infrastructure has been developed in our department. This prototype has been used as a testbed for the performance comparison among the group toolkits presented in the next section.

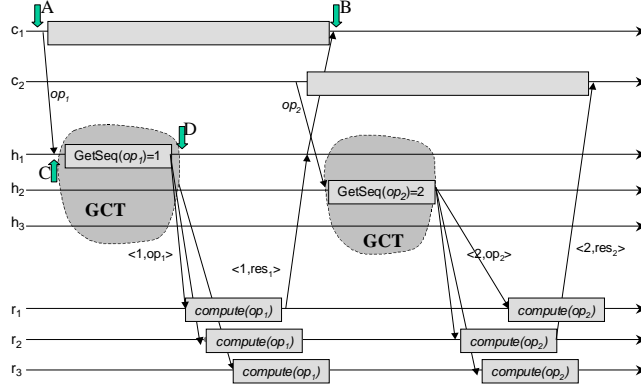


Fig. 4. A failure-free run of the 3T protocol for active software replication.

4 Performance Results

In this section we describe the performance analysis we carried out on the current IRL prototype. In particular, we first introduce the testbed platform and then explain the set of experiments done.

4.1 Testbed Platform

The testbed platform is composed by eight Intel Pentium II 600Mhz workstations that run Windows 2000 Professional as operative system. Each workstation is equipped with Java 2 Standard Edition version 1.3.0_01 ([18]) and IONA's ORBacus 4.1 for Java ([19]); also each PC is equipped with Appia ([10]), Spread ([9, 8]) and JavaGroups ([7]) group toolkit; the workstations are interconnected by a 100Mbit Switched Ethernet LAN. The replicated CORBA object used for the experiments is a simple hello-server

Params	Description	Values
#C	number of concurrent clients	1,2,4,8
#ARH	number of ARH replicas	2,4,6,8
#R	number of replicas	2,4,6,8

Table 2. Experimental Parameters

that accepts requests and immediately replies. We measured the client latency and the DSS latency varying the number of replicas (#R), the number of clients invoking the replicated server (#C) and the number of the ARH components (#ARH). As depicted in Figure 4, the client latency is the time elapsed between points A and B, whereas the DSS latency is the time elapsed between points C and D. Table 2 summarizes the experimental parameters and the ranges in which they vary.

Table 3 shows the group toolkit configuration used during the experiments i.e., the relative deployment of core and API, the total order protocol used and its type³.

Experiment 1. In this experiment we evaluated the scalability of 3T architecture in terms of the number of replicas. Therefore we set #C=1, #ARH=2 (minimum fault tolerance), and varied #R in {2, 4, 6, 8}, measuring the corresponding client and DSS latency. The experimental results are depicted in Figure 5. The client and the DSS latency

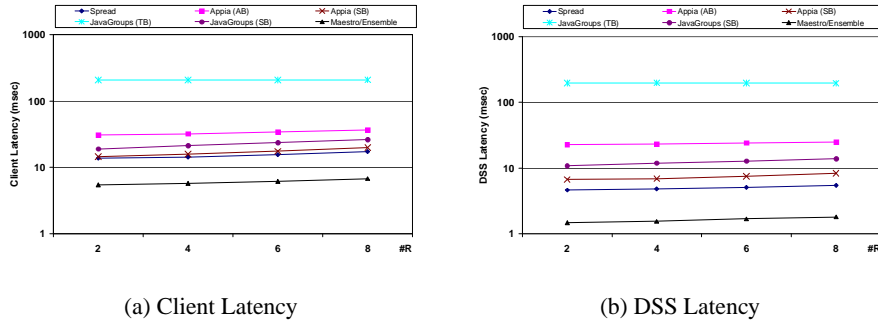
³ Due to the lack of space, we won't enter the details of the total order algorithms, which can be found in literature. We will rather focus on differences among protocols causing performance gaps while describing experimental results. Interested readers can refer to [20] for a nice survey on total order algorithms.

GCT	API and Core co-location	Protocol Name	Protocol Type
Spread	same host	Ring	token based
Appia (AB)	same process	TotalABcast	based on Skeen protocol [21]
Appia (SB)	same process	TotalSequencer	sequencer based
JavaGroups (TB)	same process	TOTAL_TOKEN	token based
JavaGroups (SB)	same process	TOTAL	sequencer based
Maestro/Ensemble	same process	Seqbb	sequencer based

Table 3. Group Toolkits Configurations.

are independent of the value of #R for any GCT. This implies that the 3T architecture is scalable with respect to the number of replicas⁴.

On the other hand, the selection of a particular GCT has an important role in the client latency. In fact the latency introduced by the DSS falls between 30% (Spread and MAESTRO/Ensemble) and 95% (JavaGroups (TB)) of that experienced by the client. From Figure 5 it can be devised that different implementations of the same protocol



(a) Client Latency

(b) DSS Latency

Fig. 5. Client and DSS Latency as a Function of #R (#C=1, #ARH=2)

(namely the total order primitive) can yield huge differences in the performance of the toolkit. More specifically, JavaGroups (TB) gets total ordering through the circulation of a token (as in TOTEM [5]) among the members of the group. This of course introduces an additional latency, not present in JavaGroups (SB) which is based on the notion of a sequencer i.e., each invocation of a total order primitive is redirected to a coordinator of the group that orders them. Also Appia shows different performance when using Appia (AB) or Appia (SB) to get total order. While Appia (SB) is based on a sequencer similar to JavaGroups (SB), Appia (AB) follows a two phase protocol⁵. Finally, the results also confirms that toolkits with a non-Java core has better performances. In particular, Spread gives better performances than the two other Java toolkits, despite of the collocation of its API and core in distinct processes.

Experiment 2. In this experiment we evaluated the scalability of the 3T architecture in terms of the number of ARH (DSS) components (and then of the group toolkit); to this end, we measured client and DSS latency as a function of #ARH, and thus we set #C=1, #R=2 (minimum fault tolerance), and varied #ARH in {2, 4, 6, 8}.

As depicted in Figure 6(a) and Figure 6(b), the client latency has the same behavior of the latency introduced by the DSS component.

⁴ The scalability of the architecture with respect to the number of replicas is a consequence of the replication handling protocol run by the middle-tier, which awaits only for the *first* reply before sending it to the client.

⁵ The sender of a total order multicast message first gathers information concerning the messages delivered by other processes (first phase), then it multicasts the message by piggybacking control information (devised by the information received in the first phase) which allows other processes to totally order it with respect to the other concurrent messages.

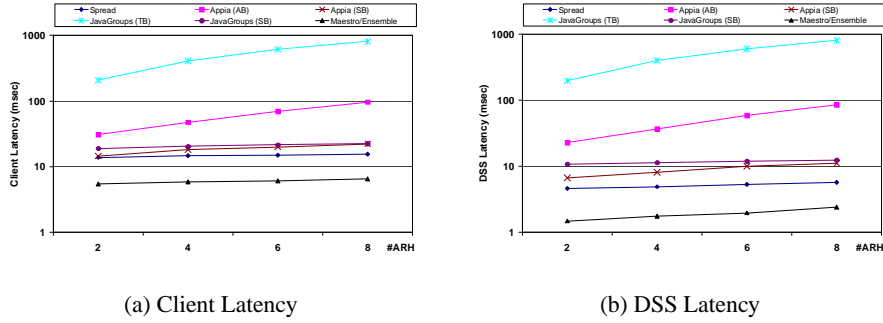


Fig. 6. Client and DSS Latency as a Function of #ARH (#C=1, #R=2)

Appia (AB) and JavaGroups (TB) don't scale well implementing protocols that have a time complexity that notably grows with the number of ARH. More precisely the token implementing JavaGroups (TB) have to touch all ARH replicas before delivering a message while Appia (AB) has to wait in the first phase replies from all the ARHs.

Experiment 3. In this experiment we evaluated the client and DSS latency as a function of the number of concurrent clients. Therefore we set #R=2 (minimum fault tolerance), #ARH=4, and varied #C in {1, 2, 4, 8}.

Figure 7 points out the experimental results. The client latency grows almost as the DSS latency does until the number of clients reaches 4; then it roughly doubles, while the DSS latency continues to increase smoothly. This is due to the additional synchronization required within each ARH replica to serve multiple concurrent clients. With 8 clients, indeed, each ARH replica receives requests from two clients, and the access to the DSS component is serialized within each ARH. As a consequence, the GCT works in the same conditions as there were only 4 clients. The little growth of the DSS latency is due to the small growth of network traffic due to the additional replicas, and it is almost independent from the toolkit, as confirmed in Figure 7(b). In contrast, synchronization within each ARH causes the doubling of the client latency with respect to that observed with 4 clients. Therefore, overall performances are mainly influenced by the GCT when the number of clients is less than or equal to the number of ARH replicas. Figure 7(b) also confirm the gap between JavaGroups (SB) and other toolkits with a non-Java core.

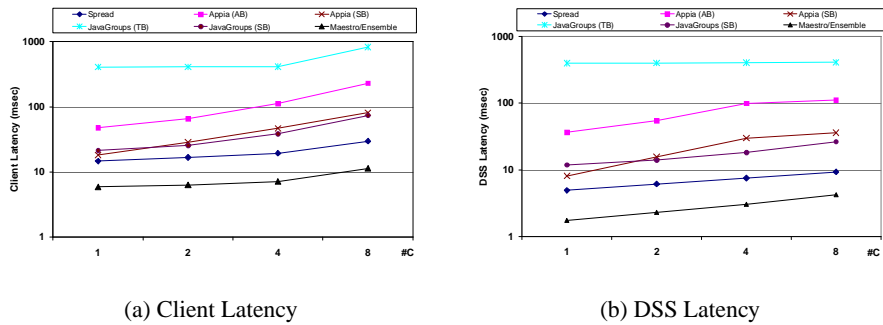


Fig. 7. Client and DSS Latency as a Function of #C (#ARH=4, #R=2)

Let us finally remark that in all the experiments Maestro/Ensemble outperforms Java implementations, which lay on a virtual machine. However, pure Java GCTs like Appia (SB) and JavaGroups (SB) can be configured to perform close to Spread, which due to the daemon-based design is comparable with Maestro/Ensemble in terms of efficiency.

5 Conclusions

In this paper we have first identified some architectural properties that influence the performance of a GCT, and then we have compared the performances of three Java toolkits and a C++/OCAML toolkit, in the context of a three-tier software replication architecture. Results are promising, as the expected performance degradation due to Java can be considered acceptable with respect to all the advantages due to the use of this language. However, a gap with C++ still exists, and optimizations are required in order to reduce it.

References

1. Schneider, F.B.: Replication Management Using the State Machine Approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993)
2. Budhiraja, N., Schneider, F., Toueg, S., Marzullo, K.: The Primary-Backup Approach. In Mullender, S., ed.: Distributed Systems. ACM Press - Addison Wesley (1993) 199–216
3. Gifford, D.: Weighted voting for replicated data. In: Proceedings of 7th ACM Symposium on Operating System Principles. (1979) 150–162
4. Birman, K., van Renesse, R.: Reliable Distributed Computing With The ISIS Toolkit. IEEE Computer Society Press, Los Alamitos (1993)
5. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., Lingley-Papadopoulos, C.A., Archambault, T.P.: The Totem System. In: Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing, Pasadena, CA (1995) 61–66
6. Vaysburd, A., Birman, K.P.: The Maestro Approach to Building Reliable Interoperable Distributed Applications with Multiple Execution Styles. *Theory and Practice of Object Systems* 4 (1998) 73–80
7. JavaGroups Web Site: (<http://www.javagroups.com>)
8. Spread Web Site: (<http://www.spread.org>)
9. Amir, Y., Stanton, J.: The spread wide area group communication system. Technical Report CNDS 98-4 (1998)
10. Appia Web Site: (<http://appia.di.fc.ul.pt/>)
11. Ensemble Web Site: (<http://www.cs.cornel.edu/info/projects/ensemble>)
12. Herlihy, M., Wing, J.: Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems* 12 (1990) 463–492
13. Marchetti, C.: A Three-tier Architecture for Active Software Replication. PhD thesis, Dipartimento di Informatica e Sistemistica, Università degli Studi di Roma “La Sapienza” (2002)
14. Baldoni, R., Marchetti, C., Tucci-Piergiorgio, S.: Asynchronous active replication in three-tier distributed systems. In: to appear in Proceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC2002), Tsukuba, Japan (2002)
15. Baldoni, R., Marchetti, C., Termini, A.: Active software replication through a three-tier approach. In: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS’02), Osaka, Japan (2002) 109–118
16. Baldoni, R., Marchetti, C., Tucci-Piergiorgio, S.: Fault-tolerant Sequencer: Specification and an Implementation. In Ezhilchelvan, P., Romanovsky, A., eds.: *Concurrency in Dependable Computing*. Kluwer Academic Press (2002)
17. IRL Project Web Site: (<http://www.dis.uniroma1.it/~irl>)
18. Java Sun: (<http://java.sun.com>)
19. IONA Web Site: (<http://www.iona.com>)
20. Defago, X.: Agreement-Related Problems: From SemiPassive Replication to Totally Ordered Broadcast. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland (2002) PhD thesis no. 2229.
21. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. *ACM Transactions on Computer Systems (TOCS)* 5 (1987) 47–76