An evaluation of the performance impact of generic group communication APIs

L. Sales*, H. Teófilo, N.C. Mendonça, J. D’Orleans, R. Barbosa and F. Trinta

Mestrado em Informática Aplicada,
Universidade de Fortaleza,
Av. Washington Soares, 1321,
CEP 60811-905 Fortaleza – CE, Brazil
Fax: +55 85 3477 3061
E-mail: leandro.shp@gmail.com E-mail: henriquetft@gmail.com
E-mail: nabor@unifor.br E-mail: jonathan.dorleans@gmail.com
E-mail: bgrafrael@gmail.com E-mail: trinta@unifor.br
*Corresponding author

Abstract: This paper presents an evaluation of the performance impact of two generic group communication APIs, namely Hedera and jGCS, over three well-known group communication systems, namely JGroups, Spread and Appia. The evaluation compared the performance of different configurations of the three group communication systems in a local clustered environment, under different message and cluster sizes, both in standalone mode and when used as plug-ins for the two generic APIs. The results show that there are significant differences in the overhead imposed by each generic API with respect to the performance of the three group communication systems, when used in standalone mode, and that those differences are strongly related to variations in message and also to the way the generic APIs and their plug-in mechanisms are implemented. Based on those results, the paper discusses some of the circumstances upon which it would be worth implementing group communication using the investigated systems.

Keywords: group communication; performance evaluation; generic APIs; high performance systems architecture.


Biographical notes: Leandro Sales is a Research Assistant at the Center of Technological Sciences, University of Fortaleza, Brazil. His main research areas are distributed systems and software engineering. He received his MSc in Applied Informatics from University of Fortaleza, Brazil.

Henrique Teófilo is a BSc student in Computer Science at the University of Fortaleza, Brazil. His main research areas are distributed systems and cloud computing.

Nabor C. Mendonça is a Titular Professor at the Center of Technological Sciences, University of Fortaleza, Brazil. His main research areas are distributed systems, software engineering and cloud computing. He received his PhD in Computing from Imperial College London, UK.

Jonathan D’Orleans is a BSc student in Computer Science at the University of Fortaleza, Brazil. His main research areas are distributed systems and artificial intelligence.

Rafael Barbosa is an Auxiliary Professor at the Center of Technological Sciences, University of Fortaleza, Brazil. His main research areas are distributed systems and computer graphics. He received his MSc in Applied Informatics from University of Fortaleza, Brazil.

Fernando Trinta is an Adjunct Professor at the Center of Technological Sciences, University of Fortaleza, Brazil. His main research areas are distributed systems, software engineering and mobile and pervasive computing. He received his PhD in Computer Science from Federal University of Pernambuco, Brazil.
1 Introduction

Group communication plays an important role in the design of fault-tolerant distributed systems (Coulouris et al., 2005). Classical group communication applications include replication, support for distributed and clustered processing, distributed transactions, resource allocation, load balancing, system management and monitoring, and highly available services (Chockler et al., 2001).

However, implementing a fully-edged group communication system (GCS) from scratch can be a daunting (and therefore error-prone) task. To overcome this situation, distributed systems researchers and tool developers have created a number of reusable GCSs, providing a variety of group communication primitives and protocols that can be used as powerful building blocks for the development of reliable distributed applications. Some of the most popular GCSs currently in use are JGroups (Ban, 1998), Spread (Amir et al., 2000), Appia (Miranda et al., 2001), and NeEM (Pereira et al., 2003).

While developers can certainly benefit from such a diversity of GCSs, for example by choosing a solution that best suits the needs and constraints of their application, they also face a new challenge: which GCS to use? Choosing an appropriate GCS for a distributed application is an important design decision that can be made difficult by the fact that those systems tend to vary widely in terms of the features they provide, including communication abstractions, quality-of-service (QoS) guarantees and delivery semantics (Chockler et al., 2001). In addition, once a developer commits to a particular GCS implementation, the distributed application source code becomes tightly coupled to that system’s API. This level of coupling is undesirable in a distributed application since it requires changes to the application code every time the chosen GCS’s API evolves. Even worse, it may discourage developers from experimenting with new GCSs in future versions of their application.

An interesting solution to decouple a given distributed application from a specific GCS implementation is to rely on generic APIs, such as those offered by Hedera (2008), jGCS (Carvalho et al., 2006) and Shoal (2008). Each of those systems provides a common programming interface and a plug-in mechanism that allows that common interface to be easily (re)implemented using the services of different existing GCSs. The use of a generic group communication API is also attractive from a performance perspective, as it frees developers to switch to the fastest GCS plug-in available, without the need to change their application code.

Even though there is an extensive body of work on the performance of individual GCSs in the literature (Abdellatif et al., 2004; Amir et al., 2004; Baldoni et al., 2002), some fundamental questions regarding the use generic APIs are yet to be fully explored. For instance, how those generic APIs impact the performance of the different GCSs they encapsulate? How is that impact influenced by factors such as message size, group size and transport protocol? Apart from the clear software engineering benefits, would there be any performance gain in replacing the services of a given GCS for those provided by a generic API? Clearly, this kind of knowledge would be of great value to distributed application developers, who could decide more confidently about when it is more appropriate (or mandatory) to use a particular GCS, and when it would be worth migrating to a generic API.

In our previous work (Sales et al., 2009), we have started to shed some light on some of the above questions by presenting the results of an initial study where we have evaluated the performance impact of two generic group communication APIs, namely Hedera (2008) and jGCS (Carvalho et al., 2006), over two GCSs, namely JGroups (Ban, 1998) and Spread (Amir et al., 2000), in a single clustered environment. In this paper, we expand upon that work by:

1 including a new GCS in the study, namely Appia (Miranda et al., 2001)
2 conducting the performance evaluation experiments in clusters of varying sizes
3 correlating the results of the three GCSs evaluated in order to investigate whether it would be worth migrating from a standalone group communication implementation to a faster solution using a generic API.

In essence, our results show that there are significant differences in the overhead imposed by each generic API with respect to the performance of the three GCSs, when used in standalone mode, and that those differences are strongly related to variations in message size and to the way the generic APIs and their plug-in mechanisms are implemented. We also have found that, for smaller messages, in the order of tens of bytes in size, migrating from JGroups or Appia to Spread using a lightweight generic API, such as that provided by jGCS, may be a viable alternative with clear performance and software engineering gains to the target distributed application.

The rest of the paper is organised as follows. Section 2 gives a quick overview of the three GCSs and the two generic APIs investigated. Sections 3 and 4 describe our evaluation method and results, respectively. Section 5 discusses the implications and limitations of our study. Section 6 covers related work. Finally, Section 7 concludes the paper and outlines our future research directions.

2 Systems and APIs evaluated

2.1 JGroups

JGroups is an open source reliable group communication toolkit written entirely in Java (Ban, 1998). It offers a high level communication abstraction, called Channel, which works like a group communication socket through which applications can send and receive messages to/from a
process group. With this abstraction, the different protocol implementations can be used by JGroups become totally transparent to application developers, who can reuse their application code across different communication scenarios and network configurations, just by reconfiguring JGroups’s underlying protocol stack.

One of the most powerful features of JGroups is that it allows developers to write their own protocol stack, by combining different protocols for message transport (for instance, TCP or UDP over IP Multicast), fragmentation, reliability, failure detection, membership control, etc. This flexibility has made JGroups particularly popular amongst middleware developers, with the system having been used to implement the clustering solution for a number of open source JEE applications servers, including JOnAS (2008) and JBoss (2008).

In our study, we used the JGroups toolkit version 2.6.10, released on 28 April 2009. JGroups is available at http://www.jgroups.org.

2.2 Spread

Spread is another open source group communication toolkit that provides a high performance messaging service that is resilient to faults across local and wide area networks (Amir et al., 2000). It offers a range of reliability, ordering and stability guarantees for message delivery. Spread is aimed at improved scalability and performance, and implements a rich fault model that supports process crashes recoveries and network partitions and merges under the extended and standard virtual synchrony semantics (Amir et al., 2000).

Spread adopts a client-server architecture, where the server (called a Spread daemon) is responsible for handling all communication amongst group members. Spread can be configured to use a single daemon in the network or to use one daemon in every computer node running group communication applications. Its server-based communication architecture avoids having heavyweight group communication protocols, like membership management, message ordering and flow control, running on all group nodes.

Although the Spread server component is written in C, the system provides native APIs for a number of different programming languages, including C++, C#, Java, Perl, Python and Ruby. It also supports cross-platform operation between Unix/Linux and Windows.

In our study, we used the Spread toolkit version 4.0, released on 4 December 2006 (the latest version available at the time of the study). Spread is available at http://www.spread.org.

2.3 Appia

Appia is an open source layered communication framework implemented in Java (Miranda et al., 2001). Like JGroups, it provides extended configuration and programming capabilities. The Appia toolkit is composed by a core that is used to compose protocols and a set of protocols that provide group communication, ordering guarantees, atomic broadcast, message fragmentation, failure detection, among other properties. In addition, Appia users can implement their own protocols and compose them with the system’s built-in protocols when creating a new protocol stack.

Even though Appia implements a fairly complete set of group communication services, it was originally developed as a more general interprocess communication system and therefore does not provide a group-specific communication API. This characteristic can make it harder for application developers to configure and use Appia’s group communication primitives, when compared with the higher-level APIs provided by other GCSs such as JGroups and Spread.

In our study, we used the Appia toolkit version 4.1.0, released on 18 January 2009. Appia is available at http://appia.di.fc.ul.pt/.

2.4 Hedera

Hedera is an open-source Java framework designed to provide a uniform API to different group communication toolkits (Hedera, 2008). Although it can be used in different application scenarios and network configurations, Hedera was originally targeted at reliable group communication within clustered environments.

Hedera has been used by the Sequoia project (Sequoia, 2008) as its group communication layer. Sequoia is a transparent middleware solution offering clustering, load balancing and failover services for replicated databases, originally developed as a continuation of C-JDBC (2008).

In our study, we used Hedera’s plug-ins for JGroups, Spread and Appia distributed with version 2.0, released on 31 October 2008. Hedera is available at http://hedera.continuent.org.

2.5 jGCS

jGCS is another generic group communication toolkit written in Java, that offers a common API to several existing GCSs (Carvalho et al., 2006). jGCS can be used by distributed applications with different group communication needs, from simple IP multicast to virtual synchrony or atomic broadcast. The architecture of jGCS relies on the inversion of control design pattern (Fowler, 2005) to decouple service implementation from service use, thus allowing the same API to be used to access the services of different GCSs. The actual service implementation that is used by jGCS is defined at configuration time.

jGCS has been originally developed within the context of the GORDA project (GORDA, 2008), which, like Sequoia, also aims at providing solutions for transparent database replication, but with a particular focus on large-scale systems.

In our study, we used jGCS’s plug-ins for jGroups, Spread and Appia distributed with version 0.6.1, released on October 29, 2007. jGCS is available at http://jgcs.sourceforge.net/.
3 Evaluation method

Our evaluation was carried out in a clustered environment composed of six computer nodes connected through a dedicated 10/100 Mbps Fast Ethernet switch. Each node had the following configuration: Linux Debian 5.0 (2.6.26-2-686 kernel) operating system; Intel Pentium IV (3.00 GHz) processor; 2 GB (DDR2) RAM; and SUN’s Java Virtual Machine version 1.6.0 12 (executed in server-side mode).

This environment was configured to emulate typical group communication scenarios used by clustered JEE application servers. In those scenarios, where the main aim is to provide client applications with fundamental QoS benefits such as high availability and scalability, group members are expected to be relatively stable and organised in small clusters within the same network domain (Lodi et al., 2007). We chose this particular application domain because it is representative of the way most GCSs are used in real world distributed applications.

To run our experiments, we used an extended version of the Java application developed by the JGroups (2007) team for their performance tests. Our extension consisted of modifying the application source code so that it could also work with the other toolkits considered in our study, i.e., Spread (through its provided Java API), Appia, Hedera and jGCS.

In each experiment, our test application was executed concurrently at each cluster node, with each node being configured to multicast 1,000 messages of equal size to all nodes in the cluster, including its own local node. This n-to-n group communication pattern is similar to the scenario in which all clustered JEE application server replicas are broadcasting their state changes to the other replicas in the cluster (Lodi et al., 2007).

We ran different sets of experiments for JGroups, Spread and Appia, both in standalone mode and when used as plug-ins for Hedera and jGCS, respectively, using clusters of two, four and six nodes in size. Each experiment used a different message size, with sizes being defined according to the following range: 10, 100, 1,000 and 10,000 bytes. These values were chosen so that we could observe the behaviour of the five systems under a broad range of message sizes.

Since JGroups can be configured to use different transport protocols, we evaluated two JGroups configurations: one using UDP over IP multicast and the other using TCP.

In both configurations we used JGroups’ SEQUENCER total order protocol, with message fragmentation and bundle disabled. The other configuration parameters were defined according to JGroups’s (2007) test configuration parameters.

In the case of Spread and Appia, we evaluated a single configuration, using TCP as the main transport protocol and their own total order protocol. Although Appia also offers a protocol stack based on UDP, it does support guarantees such as reliability and total order. Spread, on the other hand, implements a fixed protocol stack using TCP for communication amongst its server components and UDP for communication between the client applications and the server. The Spread configuration used in our experiments had a fully decentralised architecture, with one Spread server co-located with our test application in every cluster node.

Finally, we used message delivery rate as our performance metric (Jain, 1991). In our study, this metric was computed by calculating the average number of messages delivered per second at each node, at each experiment. To achieve a confidence level of 95% in our results, each experiment was repeated at least 30 times, with extreme outliers being removed using the boxplots method (Triola, 1997).

In all subsequent figures shown in Section 4, confidence intervals are represented as a pair of upper and lower lines drawn around the top of each performance bar.

4 Results

We first show the performance impact observed for the two generic APIs over each of the three GCSs investigated. We then correlate those results with the way messages are transmitted at the transport layer using each generic API, as a way to explain their performance differences.

Moreover, because Spread is known to outperforms JGroups and Appia in some communication scenarios by a significant margin (Baldoni et al., 2002), we also compare the performance results observed with Appia and the two JGroups configurations, in standalone mode, against the performance of Spread as a plug-in for both Hedera and jGCS. With this analysis we aim at investigating whether it would be worth migrating from a stand alone solution based on JGroups or Appia to a generic solution based on Spread.

4.1 Performance impact over JGroups

Figure 1 and Figure 2 show the average message delivery rates observed for the UDP and TCP JGroups configurations, respectively, in a four-node cluster, both in standalone mode and as plug-ins for Hedera and jGCS, across all message sizes.

As we can see from those two figures, for messages up to 100 bytes in size the impact of jGCS over JGroups, in either configuration, is relatively small, with jGCS delivering about 10–12% less messages on average than JGroups in standalone mode. One notable exception was observed for the UDP configuration with 1000 byte messages, where jGCS’s delivery rate is about 20% lower than that of JGroups in standalone mode. We attribute this slightly higher impact of jGCS over JGroups to a greater variation in the results observed for this particular message size, as evidenced from its wider confidence interval. Hedera, on the other hand, imposes a huge impact over the performance of the two JGroups configurations for the same range of message sizes, with the former delivering about 45–60% less messages than JGroups in standalone mode.
These results also indicate that the two generic APIs appear to be generating a constant overhead per message, independently of message size.

**Figure 1** Results for JGroups-UDP in a four node cluster

For greater messages (in the order of thousands bytes) we observe a steep performance drop for the three systems, with their delivery rate differences falling to less than 12% (TCP configuration) or 28% (UDP configuration). Note that the drop is even steeper for the TCP configuration, with the delivery rates of the three systems rapidly falling below the 500 messages per second mark. We attribute these results to the fact that, for larger messages, the increasing traffic overhead generated by JGroups (due to the increasing message fragmentation happening at the transport and network layers) starts to dominate the processing overhead imposed by the two generic APIs at the application layer.

Given the steep performance drops observed across all systems for larger messages, we have also investigated how the two JGroups configurations would scale for 10 Kbyte messages as the cluster size increases from two to four and six nodes. The results are shown in Figure 3 and Figure 4, respectively. From those figures we can see that the UDP configuration offers better results, with its transmission rates being practically unaltered across the three cluster sizes. One notable exception is the result for Hedera with the UDP configuration, whose transmission rate increases slightly as the cluster size grows. We believe this increase is not significant and fits within the margin of variation observed in most of our performance measurements. The TCP configuration, on the other hand, does not scale so well, with its transmission rates dropping rapidly as the cluster size grows.

**Figure 3** Results for JGroups-UDP with 10 Kb messages

**Figure 4** Results for JGroups-TCP with 10 Kb messages

4.2 **Performance impact over Spread**

Figure 5 shows the average message delivery rates observed for Spread, in standalone mode, and for its Hedera and jGCS plug-ins, in a four node cluster, across all message sizes.

**Figure 5** Results for Spread in a four node cluster

As with the two JGroups configurations, we can observe that jGCS’s impact over Spread is considerably smaller than that of Hedera, with the jGCS Spread plug-in delivering between 15–30% less messages than Spread in standalone mode for messages up to 1,000 bytes in size. Hedera’s
performance is even worse this time, with its Spread plug-in delivering between 80–85% less messages than Spread in standalone mode for 10–100 byte messages, and about 45% less messages than Spread for 1,000 byte messages.

On the other hand, Spread was found to be relatively less capable than JGroups for delivering larger messages, with its performance dropping rapidly (either in standalone mode or as a plug-in for Hedera or jGCS) as the message size approaches 1,000 bytes. For 10,000 byte messages, the performance of Spread drops even further in all three modes, with the impact caused by the two generic APIs being completely dominated by the network overhead. We could not correlate those results with those reported for Spread in Baldoni et al. (2002), since in that study Spread and other CGSs were evaluated in the context a simple replicated CORBA component, with the authors providing no information regarding the size of the messages being transmitted at the group communication layer.

Nevertheless, our results with Spread corroborate the results observed with JGroups that the two generic APIs seem to be generating a constant overhead per message. This means that the impact imposed by the generic APIs may not be dependent on a particular plug-in implementation and so is likely to apply to other GCSs, as we will see in the next subsection.

In terms of the cluster size impact, Spread has turned out to scale even worse than the TCP configuration of JGroups when delivering 10 Kbyte messages. As we can see from Figure 6, the performances of Spread and its two generic plug-ins drop abruptly to below the 300 messages per second mark as soon as the cluster size increases from two to four nodes, remaining at that level when the cluster grows to six nodes.

Figure 6 Results for Spread with 10 Kb messages

4.3 Performance impact over Appia

Figure 7 shows the average message delivery rates observed for Appia and its Hedera and jGCS plug-ins, in a four node cluster, across all message sizes. From that figure we can see that the impact of jGCS over Appia is virtually non-existent for all message sizes (the slight advantage of jGCS over Appia for 10 and 100 byte messages is within the margin of variation observed for those particular message sizes, as evidenced by their wider confidence intervals), while Hedera’s impact is substantial, delivering about 30–40% less messages than Appia in standalone mode for messages up to 1,000 bytes.

Figure 7 Results for Appia in a four node cluster

As with the other two GCSs, Appia and its generic plug-ins also suffer severe performance losses for 10 kbyte messages, with their delivering capabilities falling close to 25% of their delivery capabilities for smaller messages.

Regarding the impact of cluster size in the results, we can see from Figure 8 that Appia behaves similarly to the TCP JGroups configuration, with its performance and that of its two generic plug-ins dropping gradually as the cluster size increases.

Figure 8 Results for Appia with 10 Kb messages

Table 1 Message size overhead imposed by the Hedera plug-ins at the transport layer

<table>
<thead>
<tr>
<th>Cluster size</th>
<th>Appia</th>
<th>JGroups-UDP</th>
<th>JGroups-UDP</th>
<th>Spread</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1044</td>
<td>1042</td>
<td>1051</td>
<td>725</td>
</tr>
<tr>
<td>4</td>
<td>1180</td>
<td>1146</td>
<td>1155</td>
<td>893</td>
</tr>
<tr>
<td>6</td>
<td>1316</td>
<td>1250</td>
<td>1259</td>
<td>1061</td>
</tr>
</tbody>
</table>

4.4 Transport layer overhead

To investigate the possible cause for Hedera’s substantial performance overhead compared to that of jGCS, particularly for small messages, we have compared the size of the messages sent by the two generic APIs at the
application layer, against the size of the messages that are actually sent by their underlying GCSs at the transport layer, across all cluster sizes. Our findings were that Hedera adds a significant number of extra bytes to every message at the transport layer, while no such overhead exists for jGCS. The overhead numbers observed for Hedera are presented in Table 1.

From that table, it is clear that the Hedera plug-ins always send messages at the transport layer that are considerably larger than the messages originally sent at the application layer. In fact, Hedera always adds between 725 and 1,316 extra bytes to every application message, depending on the number of nodes (group members) in the cluster and the actual plug-in being used. This means that Hedera may in fact send to group members messages that are up to two orders of magnitude larger than the original messages created by the application, which explains its poor performance for smaller messages (in the order of a few bytes in size).

A further inspection of the Hedera source code revealed that those extra bytes added by Hedera to every message are actually control data generated by its plug-in mechanism to store, amongst other information, the group members’ IDs. That extra data is needed by Hedera to allow developers to define application-specific message headers, and also to send messages to a selected subset of the members of a given group. Since jGCS implements none of those features, it does not incur in any message size overhead at the transport layer. Therefore, Hedera offers a compromise between performance and flexibility, while jGCS trades flexibility for a more compact message size and, consequently, better performance.

4.5 API migration impact

As we have stated earlier, because Spread can offer better performance than JGroups and Appia in some circumstances, particularly for small messages, we were also interested in investigating the conditions upon which it would be worth migrating from a standalone solution based on Appia or JGroups to a generic API plug-in based on Spread. In other words, we want to investigate whether Spread’s improved performance would be enough to compensate for the processing and communication overhead imposed by the generic API over the underlying GCS.

Figure 9 and Figure 10 show the average message delivery rates observed for the jGCS and Hedera Spread plug-ins, respectively, in a four node cluster, versus those obtained with Appia and the two JGroups configurations in standalone mode.

From Figure 9 we can see that, for messages up to 100 bytes in size, the jGCS Spread plug-in can deliver about three times more messages than the two JGroups configurations, and about 4-5 times more messages than Appia. On the other hand, Figure 10 shows that, for the same range of message sizes, the Hedera Spread plug-in delivers far less messages than the two JGroups configurations, with a similar performance to that of Appia.

For 1,000 byte messages, the Hedera plug-in shows an even steeper performance loss compared to those of JGroups and Appia in standalone mode.

In essence, the above results suggest that migrating from a standalone GCS to a generic API can be an attractive alternative to improve application performance, as long as the following three conditions are met:

1. messages are small enough not to overload the underlying network (in our analysis, in the order of tens or hundreds of bytes)
2. the generic API offers a plug-in for another GCS that is faster than the original GCS used by the application
3. the performance gain provided by the new GCS is high enough to compensate for the performance overhead imposed by the generic API.

Note that the third condition is only met by the jGCS Spread plug-in, since Hedera’s performance overhead is consistently higher than the performance gain offered by Spread over Appia and the two JGroups configurations. Therefore, according to our results, a migration from Appia or JGroups, in standalone mode, to a generic API based on Spread, would only be worth with jGCS and not with Hedera.
5 Discussion

There are number of factors that should be considered by a distributed application developer when contemplating the possibility of using a generic group communication API. Perhaps the most important one is to consider whether having a loosely coupled communication architecture is a major design concern (for example, if the developer foresees the possibility of switching to or experimenting with new GCSs in the future). This decision is important because generic APIs, such as Hedera or JGCS, as we have shown along the paper, always impose extra levels of indirection between the application and the underlying GCS implementation, thus inevitably resulting in some performance loss.

Another factor that the developer must take into account is message size. In particular, based on the results reported in the previous section, and assuming similar execution and network environments, for messages up to 100 bytes, we can argue that it would be worth replacing an existing GCS (e.g., JGroups or Appia) by a lightweight generic API (such as jGCS), from a straight performance perspective, as this would make it easier for the developer to improve the application’s performance by switching to a faster GCS implementation (such as Spread). However, for message sizes in the order of thousands of bytes, the choice between using a particular GCS directly, in stand-alone mode, or indirectly, encapsulated behind a generic API, should be made based entirely on the software needs and constraints of the application at hand. The reason is that, within that message size range, network transmission costs tend to predominate over the performance overhead imposed by the generic API over the underlying GCS implementation, thus reducing the possibility of improving application performance by simply switching to a different GCS plug-in.

Despite the merits of our study, we are aware that our results are still limited in a number of ways. Above all, we have only investigated the behaviour of five group communication solutions under a local clustered environment. In this regard, we have deliberately evaluated three of the most well-known GCSs available, namely JGroups, Appia and Spread, and the only two existing generic APIs which provide plug-ins for those three systems, namely Hedera and jGCS. In addition, we have set up an experimental environment which emulates typical clustered JEE application servers, with similar characteristics to those of earlier studies reported in the literature. All these decisions increase our confidence that our experimental test bed is likely to be representative of the state-of-the-practice in many real-world group communication applications.

Some aspects of our results still require further investigation. For example, we need to determine the extent to which the performance degradation observed with larger messages is a characteristic of our test environment. In other words, is that problem independent of the GCSs and generic APIs used?

Finally, we have only compared the three GCSs and the two generic APIs from a performance standpoint. In practice, replacing one GCS for another requires a careful analysis of many other factors, such as fault-tolerance, memory footprint, and the syntactic and semantic differences between the target systems APIs. We plan to address these as well as other limitations in our future work.

6 Related work

Being three of the most popular GCSs currently available, JGroups, Appia and Spread have already been evaluated in a number of earlier performance studies reported in the literature, e.g., Abdellatif et al. (2004), Amir et al. (2004), Baldoni et al. (2002), JGroups (2007). In Abdellatif et al., (2004) and JGroups (2007), the authors compare the performance of different JGroups configurations, under a variety of network conditions. A similar study has been described for Spread by Amir et al. (2004). In contrast to those works, the primary aim of our study is not to analyse the performance of different GCSs configurations, in standalone mode, but rather to evaluate the impact that different generic APIs would impose on existing GCSs. In this way, we aim at providing relevant experimental information to help distributed application developers decide on when to use a concrete GCS implementation, such as JGroups, Appia or Spread, and when to hide such a system from the application code under a generic API, such as that provided by Hedera and jGCS.

Another work comparing the performance of several GCSs written in Java (including an earlier version of JGroups, called JavaGroups), under different usage scenarios and different network conditions, is described by Baldoni et al. (2002). However, that work is not concerned with evaluating the impact of any generic API on any particular GCS.

7 Conclusions and future work

This paper presented an evaluation of the performance impact of two generic APIs, namely Hedera and jGCS, over three well-known GCSs, namely JGroups, Appia and Spread. In essence, our results show that there are significant differences in the overhead imposed by each generic API over the performance of the three GCSs, when used in standalone mode, with Hedera offering by far the worst results. The study also shows that the performance differences observed across all systems are strongly related to variations in message sizes (for messages sizes in the order of a few thousands of bytes those differences tend to be completely dominated by the system’s increasing network overhead) and also to their inherit implementation characteristics (Hedera’s dismal performance is largely due to its significant message size overhead imposed at the transport layer).

We are currently pursuing two main research lines. The first one consists of replicating the same set of experiments
described here under a wider variety of cluster scenarios. The main idea is to investigate whether the results described in this paper scale to larger (possibly non-local) clusters. The second research line aims at evaluating the impact of migrating to a generic API specifically in the context of the clustered architecture of an existing JEE applications server, such as JBoss (2008) or JOnAS (2008). The idea, in this case, is to investigate whether the same performance gains observed in our current test environment will also occur in this new scenario, where the size of the messages exchanged between group members will vary according to the size of the session states maintained by each replicated JEE server.

References


Notes

1 A similar range of message sizes was also used in a previous evaluation of JGroups in the context of clustered J2EE application servers (Abdellatif et al., 2004).