Performance Impact Analysis of Two Generic Group Communication APIs

Leandro Sales, Henrique Teófilo, Jonathan D’Orleans, Nabor C. Mendonça, Rafael Barbosa, Fernando Trinta

Mestrado em Informática Aplicada, Universidade de Fortaleza
Av. Washington Soares, 1321, CEP 60811-905 Fortaleza – CE – Brazil
leandro@enovar.com.br, henriquetfi@gmail.com, jonathan.dorleans@gmail.com, nabor@unifor.br, bgrafael@gmail.com, trinta@unifor.br

Abstract

This paper presents an analysis of the performance impact of two generic group communication APIs, namely Hedera and jGCS. The analysis was carried out in a four-node cluster by comparing the performance of different configurations of two well-known group communication middleware systems (i.e., JGroups and Spread), under different message sizes, both in standalone mode and when used as plug-ins for the two generic APIs. The results show that there are significant variations in the performance overhead imposed by each generic API on both JGroups and Spread, and that those differences are strongly related to variations in message size and the way each generic API is implemented. The paper also discusses whether it would be worth migrating from a standalone group communication implementation to a generic API.

1. Introduction

Group communication plays an important role in the design of fault-tolerant distributed systems [8]. 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 [7].

However, implementing a fully-fledged 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 [5], Spread [3], and Appia [15].

While developers can certainly benefit from such a diversity of GCSs, for example by choosing a solution that best suits their application’s needs and constraints, 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 guarantees and delivery semantics [7]. 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 for at least two main reasons: (i) it requires changes to the application code every time the chosen GCS’s API evolves; and, most importantly, (ii) 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 [9], jGCS [6] and Shoal [16]. 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 switching 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 (e.g., [1][2][4]), 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 this work, we start to shed some light on some of the above questions. We present an analysis of the performance impact of two generic group communication APIs, namely Hedera [9] and jGCS [6], over two well-known CGSs, namely JGroups [5] and Spread [3], in a four-node cluster. In essence, our analysis results show that there are significant differences in the overhead imposed by each generic API with respect to the performance of both JGroups and Spread, when used in standalone mode, and that those differences are strongly related to variations in message size and to the way the generic APIs are implemented. We also have found that migrating from a moderately slow group communication solution (such as JGroups) to high-performance one (such as Spread), using a lightweight generic API (such as jGCS), may be a viable alternative with clear performance and software engineering gains to the target distributed application.

The rest of the paper is organized as follows. Section 2 gives a brief overview of the two 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.

2. Systems and APIs Evaluated

2.1. JGroups

JGroups is an open source reliable group communication toolkit written entirely in Java [5]. 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 that are 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 being part of the clustering solution for a number of open source JEE applications servers, including JOnAS [13] and JBoss [11].

In our study, we used JGroups version 2.6.2, released on February 26, 2008. 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 [3]. 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 [3].

Spread adopts a client-server architecture, where the server (called 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. That server-centered communication architecture avoids having heavyweight group communication protocols, like membership management, message ordering and flow control, running on all group nodes.

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

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

2.3. Hedera

Hedera is an open-source Java framework designed to provide a uniform API to different group communication toolkits [9]. Although it can be used in different application scenarios and network configurations, the system was originally targeted at reliable group communication within clustered environments [9].

In our study, we used the Hedera plug-ins for JGroups and Spread distributed with version 1.6.3,

2.4. jGCS

jGCS is another generic group communication toolkit written in Java, that offers a common API to several existing GCSs [6]. jGCS can be used by distributed applications with different group communication needs, from simple IP Multicast to reliable atomic broadcast. The architecture of jGCS decouples service implementation from service use, thus allowing the same API to be used to access the services of different GCSs. The actual service implementation used by jGCS is defined at configuration time.

In our study, we used the jGCS plug-ins for JGroups and Spread 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 four PCs connected through a dedicated 10/100 Mbps Fast Ethernet switch. Each PC was configured as follows: Linux Ubuntu 7.10 (2.6.22.14-generic kernel); Intel Pentium IV (3.00 GHz) processor; 2 GB (DDR2) RAM; and SUN’s Java Virtual Machine version 1.6.0_03 (executed in server-side mode). This environment was setup to emulate typical clustering scenarios used by JEE application servers, in which process groups are expected to be relatively stable and moderate in size [14].

In each experiment, our test application was executed at each cluster node, with each node being configured to concurrently multicast 1000 messages of equal size to all nodes in the cluster, including its own local node, which means that a total of 4000 messages were delivered at each node during each experiment. This particular group communication pattern emulates a highly-available JEE clustered scenario where each application server broadcasts its state changes to all the other servers in the cluster [14].

We ran different sets of experiments for JGroups and Spread, both in standalone mode and plugged to Hedera and jGCS, respectively, using messages of 1, 10, 100, 1000 and 10000 bytes in size. These numbers, which are similar to those used in a previous evaluation of JGroups [1], allowed us to observe the behavior of the four systems under a broad range of message sizes and network loads.

We evaluated two JGroups configurations, each one using a different transport protocol (TCP and UDP over IP Multicast, respectively). In both configurations, we used SEQUENCER as the total order protocol, with message fragmentation and bundle disabled. The other configuration parameters were defined according to JGroups’ test environment [12].

In the case of Spread, we evaluated a single configuration since it implements a fixed protocol stack. The configuration used Spread’s own total order protocol and had a fully decentralized architecture, with one Spread daemon running in each cluster node.

Finally, we used message delivery rate as our performance metric [10]. 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%, each experiment was repeated at least 30 times, with extreme outliers removed via the boxplots method [10].

4. Results

4.1. Performance Impact over JGroups

Figure 1 shows the average message delivery rates observed for the two JGroups configurations, both in standalone mode and as Hedera and jGCS plug-ins, across all five message sizes. In that figure, confidence intervals for each mean value are shown as upper and lower limits drawn around the top of their respective bar.
As we can see from Figure 1, for messages up to 100 bytes in size the impact of jGCS over JGroups, in either configuration, is relatively small, with jGCS delivering about 10-20% less messages in average than JGroups in standalone mode (one notable exception was observed for the UDP configuration with 100 byte messages, where jGCS’ delivery rate is about 40% lower than that of JGroups in standalone mode). 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 that generic API delivering about 60-70% less messages than JGroups in standalone mode.

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 30%. Note that the drop is even steeper for the UDP configuration, with the delivery rates of the three systems rapidly falling below the 2000 messages per second mark. We attribute these results to the fact that, for larger messages, the traffic overhead generated by JGroups, which is caused by the increasing message fragmentation taking place at the transport and network layers, tends to dominate any processing overhead imposed by the generic APIs at the application layer. These results also indicate that the two generic APIs are generating a constant overhead per message, independently of message size.

### 4.2. Performance Impact over Spread

Figure 2 shows the average message delivery rates observed for Spread, in standalone mode, and for its Hedera and jGCS plug-ins, across the five message sizes. Confidence intervals are shown as before.

As with the two JGroups configurations, we can observe that jGCS’s impact over Spread is non substantial for messages up to 100 bytes in size, although this time the performance differences between that generic API and its underlying GCS implementation are even smaller, with the jGCS Spread plug-in delivering about 5-10% less messages than Spread in standalone mode. As before, Hedera offered the worst results by a large margin. Specifically, the Hedera Spread plug-in delivers about 80-85% less messages than Spread in standalone mode, for 1-10 byte messages. On the other hand, Spread was found to be less scalable than JGroups for larger messages, with its performance dropping rapidly (either in standalone mode or as an API plug-in) as message sizes reach 1000 bytes. For 10000 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. These results corroborate the finding observed with JGroups that the two generic APIs are generating a constant overhead per message. This also 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.

### 4.3. Transport Layer Overhead

To investigate the possible cause for Hedera’s substantial overhead compared to that of jGCS, we have compared the size of the messages sent using the two APIs at the application layer, against the size of the messages that are actually sent through the network at the transport layer. Our finds were that Hedera always sends messages at the transport layer that are considerably larger than the messages originally sent at the application layer, while no significant transport layer overhead was observed for jGCS. In fact, Hedera always adds around between 800 and 1100 extra bytes to every message, depending on which plug-in is being used, which explains its dismal performance for small messages (up to 1000 bytes) compared with the performance of jGCS for the same range of message sizes.

A further inspection of the Hedera source code revealed that its extra bytes are used as control data, and include, amongst other information, the IDs of all group members to which the message is being transmitted. Given that jGCS also implements a similar set of generic group communication primitives without incurring in any message size overhead, we believe that an important first step towards improving Hedera’s performance would be to drastically reduce its communication overhead at the transport layer.

### 4.4. JGroups vs. jGCS/Spread

Our analysis has shown that Spread tends to outperform JGroups by a significant margin, when
used in standalone mode, particularly for small messages. This result is in accordance with previous studies reported in the literature (e.g., [4]). Here we compare the performance of the two JGroups configurations, in standalone mode, against the performance of the jGCS Spread plug-in. This analysis is meant to evaluate whether migrating from a JGroups-based standalone solution to a generic solution implemented using jGCS on top of Spread would bring any notable performance again (i.e., whether Spread’s improved performance compared with that of JGroups would compensate for any potential overhead imposed by the jGCS API).

Figure 3 shows the average message delivery rates observed for the two JGroups configurations, in standalone mode, versus the average message delivery rates observed for the jGCS Spread plug-in, across the five message sizes investigated. From that figure, we can see that the Spread plug-in implemented by jGCS can deliver 3 to 4 times more messages than the two JGroups configurations in standalone mode, for messages up to 100 bytes in size, and up to 1.7 more messages than the UDP configuration of JGroups for messages of 1000 bytes in size. As the message size approaches 10000 bytes, the two systems suffer from severe performance losses, as their implementation differences start to be dominated by the increasing network overhead.

These results show that migrating from a standalone GCS to a generic API can be an attractive alternative to improve application performance, as long as (i) messages are small enough not to overload the underlying network (in our analysis, in the order of tens or hundreds of bytes); (ii) the generic API offers a plug-in for another GCS that is faster than the original GCS used by the application; and (iii) the performance gain provided by the new GCS is high enough to compensate for the performance overhead imposed by the generic API. Note that that third condition is not met by the Hedera Spread plugin, since Hedera’s overhead is greater than the performance gain offered by Spread over the two JGroups configurations.

5. Discussion

There are number of factors that should be considered by a distributed application developer when contemplating the possibility of migrating to a generic group communication API. Perhaps the most important one is to assess whether having a loosely coupled communication architecture is a major design concern (for example, when the developer foresees the possibility of switching to or experimenting with new GCSs in the future).

Another factor that the developer must take into account is communication overhead. 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, from a straight performance perspective, it would be worth replacing an existing solution based on JGroups by a lightweight generic API, such as that provided by jGCS, as this decision 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, where network congestion starts to severely affect the performance of all group communication solutions investigated here, the choice between using a standalone solution directly, or encapsulated behind a generic API, should be made based entirely on the software needs and constraints of the application at hand.

Although we are confident on the relevance of our results, we are also aware that our study is still limited in several ways. Above all, we have only investigated the behavior of four group communication solutions under a single cluster environment. In this regard, we have deliberately selected two of the most well-known GCSs available, namely JGroups and Spread, and two existing generic APIs which provide plug-ins for those two systems, namely Hedera and jGCS. In addition, we have setup an experimental environment which emulates typical JEE clustering scenarios, as reported in the literature. All these decisions support our belief that our experimental test bed is likely to be representative of the state-of-the-practice in many real-world group communication applications.

Another limitation of our work is that we have only approached the problem 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 in their respective APIs. We plan to address those limitations in our future work.
6. Related Work

Being two of the most popular GCSs currently available, JGroups and Spread have already been used as targets for a number of performance evaluation studies (e.g., [1],[2],[4],[12]). In [1] and [12], the authors compare the performance of different JGroups configurations, under a variety of network conditions. A similar study has been described for Spread in [2]. 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 in [4].

In contrast to those works, the primary aim of our study is not to analyze the performance of different GCSs individually, but rather to evaluate the impact that a generic API, such as that provided by Hedera and jGCS, would impose on existing GCSs. In this way, our aim is to provide relevant experimental information to help distributed application developers decide on when it would be worth migrating from a standalone GCS implementation to a generic API.

7. Conclusions and Future Work

We have presented an analysis of how two generic group communication APIs, Hedera and jGCS, impact the performance of two well-known GCSs, JGroups and Spread. Our main results are: (i) Hedera’s impact is significantly higher than that of jGCS for both JGroups and Spread, particular for small messages (in the order of tens of bytes); and (ii) their performance differences tend to disappear as the message size grows over a thousand bytes, due to the increasing network traffic. We have also discussed when it would be worth migrating from a JGroups-based solution to a generic solution based on jGCS and Spread.

As future work, we plan to extend our analysis to other GCSs (e.g., Appia [15]) and generic APIs (e.g., Shoal [16]), under a wider range of clustering scenarios, to investigate the extent to which our results can be generalized. We also plan to conduct similar experiments within the context of existing JEE applications servers, such as JBoss [11] and jOnAS [13], which rely on JGroups as part of their clustering solution. The idea, in this case, is to evaluate whether the same performance differences observed in our current test environment will also occur in this new context, 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.

8. References