Development and Evaluation of a Generic Group Communication Layer

Leandro Sales[1], Henrique Teófilo[2], Nabor C. Mendonça[2]

[1] Dipartimento di Elettronica e Informazione – Politecnico di Milano
Via Ponzio, 34/5 – 20133 Milano – Italy

leandro.shp@gmail.com

[2]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321 – 60811-905 Fortaleza – CE – Brazil

henriquetft@gmail.com, nabor@unifor.br

Abstract. *Generic group communication frameworks offer several benefits to developers of clustered applications, including better software modularity and greater flexibility in selecting a particular group communication system. However, current generic frameworks only support a very limited set of group communication primitives, which has hampered their adoption by many "real-world" clustered applications that require higher-level group communication services, such as state transfer, distributed data structures and replicated method invocation. This paper describes the design, implementation and initial evaluation of G2CL, a Generic Group Communication Layer that offers a set of commonly used high-level group communication services implemented on top of an existing generic framework. Compared to current group communication solutions, GCL offers two main contributions: (i) its services can be configured to run over any group communication system supported by the underlying generic framework; and (ii) it implements the same service API used by JGroups, a popular group communication toolkit, which may reduce its learning curve and make the task of migrating to G2CL particularly attractive for JGroups users.*

1. Introduction

Group communication, i.e., the ability to reliably transmit messages amongst a group of processes, plays an important role in the design of dependable distributed system [Couloris et al. 2005]. This form of communication has been particularly valuable in clustered environments, where classical group communication applications include replication, load balancing, resources management and monitoring, and highly available services [Chockler et al. 2001].

A group communication system (GCS) implements a set of group communication services that can be reused across many different applications. Some of the most popular GCSs currently in use are JGroups [Ban 1998], Spread [Amir et al. 2000] and Appia [Miranda et al. 2001], each providing its own set of communication primitives and protocols. Choosing an appropriate GCS for a given clustered application is an important design decision that can be made difficult by the fact that those systems tend to vary widely not only in terms of the communication abstractions

they implement, but also in terms of the delivery semantics and quality-of-service (QoS) guarantees they provide [Chockler et al. 2001]. Another difficulty is that, once a developer commits to a particular GCS, her application code becomes tightly coupled to that system's API. From a software engineering perspective, this level of coupling is undesirable since it requires changing the application code every time the target API evolves; even worse, it makes it extremely hard to migrate the application to a different GCS, which may discourage developers from experimenting with new (possibly more effective) GCSs in future versions of their software.

To avoid coupling their application code to a specific GCS, developers can rely on generic group communication frameworks, such as Hedera [Hedera 2008], jGCS [Carvalho et al. 2006] and Shoal [Shoal 2008].[1] Each of those systems provides a common API for a number of existing GCSs, and a plug-in mechanism that can be used to select a particular GCS or to incorporate new GCSs. The main advantage of using a generic framework is that developers can easily switch between different GCSs (for instance, to meet new communication requirements or to accommodate changes in the application's execution environment) simply by selecting a different GCS plug-in at configuration time, without the need to change the application code.

Despite their obvious benefits, current generic frameworks only support a limited set of group communication primitives aimed at basic services such as group management (creation, change notification, etc.) and message transmission [Sales et al. 2009a, Sales et al. 2009b]. This limitation has hampered their adoption by many "real-world" clustered applications that require higher-level group communication services, such as state transfer, distributed data structures and replicated method invocation. Services of this kind are already provided by some existing GCSs, most notably JGroups [Ban 1998], whose *building blocks* have been extensively used in the development of a number of mature middleware technologies (for example, JBoss [JBoss 2009] and JOnAS [JOnAS 2009]). However, since JGroups was not developed with a generic API in mind, its high-level services are tightly-coupled to its own protocol stack, which makes them virtually impossible to reuse with a different GCS. Therefore, there is a natural need to extend existing generic frameworks with support for high-level group communication services, such as those provided by JGroups, but whose implementation is not coupled to any particular GCS implementation.

In this paper, we describe the design, implementation and initial evaluation of *G2CL*, a *G*eneric *G*roup *C*ommunication software *L*ayer that implements a set of commonly used high-level group communication services on top of an existing generic framework. Compared to current group communication alternatives, GCL offers two main benefits:

1. Its set of high-level services can be easily (re)configured to run over any GCS supported by the underlying generic framework. For example, a G2CL state transfer service implemented on top of jGCS [Carvalho et al. 2006] could be run over either Spread, Appia or JGroups, which are three of the GCSs currently supported by jGCS.

[1] In this paper, we use the term *generic* to convey the notion of greater implementation flexibility and portability.

2. It implements the same building block API used by the popular JGroups toolkit. This means that migrating an existing clustered application to G2CL should be particularly easy for JGroups users. In addition, developers not familiar with JGroups could still benefit from JGroups' extensive API documentation and code base as a guide to use G2CL, thus reducing their learning curve.

We have successfully applied G2CL to replace JGroups as the group communication mechanism in the cluster architecture of the JOnAS Java EE application server [JOnAS 2009]. The migration from JGroups to G2CL in the JOnAS source code has been done with relatively little programming effort, and has allowed us to evaluate the impact of G2CL, when configured with different GCS plug-ins, on the performance of JOnAS under a variety of load conditions. These results build our confidence that G2CL can be an effective addition to set of programming tools currently available for developers of clustered applications.

The rest of the paper is organized as follows. Section 2 gives a brief overview of two technologies that have greatly influenced our work on G2CL, namely JGroups and jGCS. Section 3 describes the main design decisions and implementation strategies used in the development of G2CL. Section 4 reports on our initial evaluation of G2CL using JOnAS as a case study. Finally, Section 5 concludes the paper and outlines our future research agenda.

2. Related Technologies

2.1. JGroups

JGroups [Ban 1998] was one of the first group communication toolkits written entirely in Java. It provides a simple API for accessing its basic group communication services, whose main component is the *Channel* interface. This interface is used to send/receive messages asynchronously to/from a group of processes, and to monitor group changes by means of the *Observer* design pattern [Gamma et al. 1995]. Currently, JGroups offers a single implementation of the *Channel* interface, called *JChannel*.[2]

The *Channel* interface hides the actual protocol stack used by JGroups for message transmission. However, JGroups allows developers to configure their own protocol stack, by combining different protocols for message transmission (for instance, TCP or UDP over IP Multicast), data fragmentation, reliability, security, failure detection, membership control, etc. This can be done via an external XML file, whose properties are loaded by JGroups at initialization time, thus avoiding the need to change the application code directly.

On top of the basic services provided by the *Channel* interface, JGroups implements another set of higher-level services, called *building blocks* [JGroups 2009], which offer more sophisticate group communication abstractions for application developers. These include services such as MessageDispatcher, which implements primitives for synchronous message transmission; RPCDispatcher, which implements a

[2]In our work, we have used JGroups version 2.6.10, released on April 28, 2009. JGroups is available at `http://www.jgroups.org`.

remote invocation mechanism for replicated objects on top of the MessageDispatcher service; and ReplicatedHashMap, which implements a distributed version of the *HashMap* class of Java on top of the RPCDispatcher service.

Due to its great flexibility in defining customized protocol stacks, and also to its rich set of building blocks, JGroups has been a popular choice amongst clustered application developers, having recently been incorporated as part of the JBoss project [JBoss 2009].

2.2. jGCS

The *Group Communication Service for Java* (jGCS) [Carvalho et al. 2006] is a generic group communication framework that aims at providing a common Java API to several existing GCSs. Its ultimate goal is to facilitate reuse of the different services implemented by those systems without requiring substantial changes in the source code of the target application.

The jGCS architecture relies on a plug-in mechanism based on the *Inversion of Control* (IoC) design pattern [Fowler 2004]. This mechanism is used by jGCS to decouple its service API from the underlying service implementation, thus allowing the same API to be reused across different GCSs. The actual service implementation (plug-in) used by jGCS can be defined at initialization time, via an external configuration file. The current version of jGCS offers plug-ins for several GCSs, including JGroups, Spread [Amir et al. 2000] and Appia [Miranda et al. 2001].[3]

The jGCS API is divided into four complementary interfaces, namely *configuration interface*, *common interface*, *data interface*, and *control interface*. These are described in more details below.

Configuration Interface This interface decouples the application code from implementation-dependent group communication concerns, such as group configuration and specification of message delivery guarantees. The actual GCS plug-in to be used is defined at configuration time, by means of an external configuration file. At execution time, the jGCS services are instantiated according to the specified configuration, using a dependency injection mechanism [Fowler 2004] or a service locator [Alur et al. 2001].

The main classes of this interface are *ProtocolFactory*, which implements the *Abstract Factory* design pattern [Gamma et al. 1995] to allow the initialization of new protocol instances based on the underlying plug-in configuration; *GroupConfiguration*, which encapsulates group information (e.g., the group ID) necessary to open a new group session through which the application can exchange messages with other group members and monitor group membership changes; and *Service*, which encapsulates the specification of message delivery guarantees to be used during message transmission.

[3]In our study, we have used jGCS version 0.6.1, released on October 29, 2007. jGCS is available at `http://jgcs.sourceforge.net/`.

Common Interface This interface contains common classes shared by all other interfaces. The main class of this interface is *Protocol*, whose instances are created by the *ProtocolFactory* classe from the configuration interface. A *Protocol* object is used to create, for a given *GroupConfiguration* object, the objects responsible for message exchange and group membership management, of types *DataSession* and *ControlSession*, respectively, described next.

Data Interface This interface contains classes responsible for sending and receiving group messages. The main classes of this interface are *DataSession*, which is used to send messages to a group and also to register *observers* [Gamma et al. 1995] to handle messages received from that same group; *Message*, which encapsulates a message to be sent or received from a group and the address of the sender; and *MessageListener*, which must be implemented by all *observers* registered with a *DataSession*.

To avoid forcing any specific data format or serialization mechanism on the application, the message body is stored as a *byte array* inside *Message*, with the application being responsible for serializing the message before transmission and deserializing it after receipt.

Control Interface This interface contains classes responsible for group management, from simple notifications of members joining or leaving a group to the creation of new virtual group views. The main classes of this interface are *ControlSession*, which provides methods for members to join or leave a group and also to register *observers* to listen to notifications of membership changes (e.g., join, leave and failure of members); *ControlListener*, which must be implemented by all *observers* registered with a *ControlSession*; *MembershipSession*, which is an extension of class *ControlSession* used to obtain a list of members currently connected to a group and also to register *observers* to listen to changes in group views; and *MembershipListener*, which must be implemented by all *observers* registered with a *MembershipSession*.

3. G2CL

G2CL is an extensible group communication software layer that sits on top of existing generic frameworks. Its main design goal is to offer a more sophisticated set of generic group communication services, similar to those provided by the building blocks of JGroups, but with all the benefits associated with the use of a loosely-coupled software architecture. In this regard, G2CL main advantage is the greater flexibility for configuring its underlying group communication mechanism, which can be any GCS supported by its underlying generic framework.

During the development of G2CL we have taken some important design decisions, discussed below.

3.1. Main Design Decisions

3.1.1. Choice of Generic Framework

Our first design decision was concerned with selecting the generic framework to be used as the basis for the implementation of G2CL. Of the three generic frameworks currently available, i.e., Hedera [Hedera 2008], jGCS [Carvalho et al. 2006] and Shoal [Shoal 2008], only Hedera and jGCS were considered mature enough for our purposes, with both providing plug-ins for several existing GCSs. Shoal, on the other hand, only provides support for a single GCS (namely, JXTA [JXTA 2008]) and thus was discarded as a possible generic framework candidate. In the end, we chose jGCS over Hedera because the former implements a well-designed API, based on sound object-oriented design principles and patterns, and also because it imposes a much lower performance overhead compared to the overhead observed for Hedera, particularly for small messages [Sales et al. 2009a, Sales et al. 2009b].

3.1.2. Service Implementation Model

Another important design decision was concerned with defining an appropriate implementation model for G2CL. Given the rich set of group communication building blocks offered by JGroups, and its popularity amongst clustered application developers, we have decided to implement the G2CL services following, whenever possible, the same building block API (including class names and method signatures) used by JGroups. As we have explained previously, this decision has the potential to facilitate the task of migrating an existing clustered application based on JGroups to G2CL, since both systems implement similar APIs. Another benefit is that G2CL users could greatly reduce their learning curve by leveraging on JGroups' extensive API documentation and code base.

3.1.3. jGCS Extensions

During the design of G2CL we have identified the need to make some minor extensions to the classes and interfaces originally provided by jGCS. These extensions are described below.

As it is typical with other communication abstractions that encapsulate lower-level services, to implement some of the G2CL services we needed a way to add service-specific headers to application messages in a manner that is separate from their body. Such headers would be used to store control information relevant to the implementation of some services, but which could not be exposed to the application. Since this facility is not readily supported by the *Message* class currently provided by jGCS, we had to define a new message class, called *G2CLMessage*.

In order to maintain compatibility with the *DataSession* classe of jGCS, *G2CLMessage* implements jGCS's *Message* interface. This allows *G2CLMessage* objects to be transmitted as any other message using any jGCS plug-in.

Another extension made to jGCS was the implementation of a

new *DataSession* class, called *MarshalDataSession*, which works like an *adapter* [Gamma et al. 1995] between the G2CL services and the original *DataSession* used by the jGCS plug-ins. The main responsibility of this new class is to intercept all message transmission calls made to the plug-in by the application and then execute the necessary transformations to convert between a message of type *G2CLMessage* and another message of type *Message*. In this way, all G2CL services must rely only on *MarshalDataSession* for message transmission (instead of the original *DataSession* class of jGCS).

3.2. Implemented Services

The initial set of group communication services implemented as part of G2CL was selected based on an informal analysis of the JGroups services that are most commonly used in practice. The selected services were classified into two groups, named *high-level services* and *service decorators*, described below.

3.2.1. High-level Services

These services encapsulate a *MarshalDataSession* instance by hiding its basic message transmission functionality, so as to provide application developers with a more sophisticated group communication API. Four services were initially implemented as part of this group: *MessageDispatcher*, *RpcDispatcher*, *ReplicatedHashMap* and *StateTransferDataSession*.

Due to space limitations, and because those services provide the same set of functionalities provided by their corresponding services in JGroups, with a similar API, we will omit the details of their implementation from the paper. For a more detailed account of those services, including their semantics and APIs, the interested reader is referred to [JGroups 2009].

3.2.2. Service Decorators

Services of this group add extra functionalities (such as message fragmentation and encryption) to the basic message transmission service provided by the *MarshalDataSession* class. As the group name implies, these services are based on the *Decorator* design pattern [Gamma et al. 1995]. Their implementation keeps the same interface provided by *MarshalDataSession*, so that their use is completely transparent to the application.

Currently, G2CL provides four service decorators, namely *FragDataSession*, *BundleDataSession*, *CompressDataSession* and *CryptoDataSession*. These services provide mechanisms for message fragmentation, message bundle, message compression and message encryption, respectively.

Each service decorator can be used either in isolation, or combined with other service decorators, forming a *chain of responsibility* [Gamma et al. 1995] where different decorators can be added or removed from the chain without affecting the application code.

To facilitate the use and configuration of service decorators, G2CL provides a *MarshalDataSessionFactory* class whose main responsibility is to creat a new *MarshalDataSession* instance. If necessary, the *MarshalDataSessionFactory* can also instantiate a chain of decorators for the new *MarshalDataSession* object.

The creation of both the *MarshalDataSession* instance and its chain of decorators can be configured by the user in a manner that is independent of the application code, using a dependency injection mechanism or a service locator.

4. Evaluation

To assess the migration effort and potential performance impact associated with the use of G2CL in a real clustered application, we have conducted a case study involving the JOnAS Java EE application server [JOnAS 2009].

The reason for selecting JOnAS as our target application is two-fold: (i) it is a mature clustered technology of non-trivial size (in the order of 230.000 lines of Java code); and (ii) it makes intensive use of a number of group communication services and building blocks provided by JGroups, which have similar services already implemented as part of G2CL.

4.1. JOnAS Overview

The *Java Open Application Server* (JOnAS) is an open source implementation of the Java EE 5 specification [SUN 2006]. This specification includes a number of Java-based technologies, such as EJB, JMS, Servlets, JSP, JSF, JPA, JAX-WS and JAX-RPC, all of which are fully supported by JOnAS.[4]

JOnAS supports the creation of reliable EJB applications by providing a high-availability (HA) service based on a cluster of JOnAS instances. When a client application requests the creation of a *Stateful Session Bean* (SFSB) component, one of the servers in the cluster is chosen to respond to that client's invocations until the client requests the removal of that SFSB. Before sending a response to the client, the server propagates any change in the state of the SFSB to the other servers in the cluster, which act as backup servers for that component. If the server initially allocated to a replicated component fails, the state of the SFSB can be recovered by one of its backup servers, which will start handling future invocations for that component on behalf of the failed server.

To implement its HA service JOnAS relies on a RMI-like replication protocol called *Clustered Method Invocation* (CMI), which is specifically tailored for transparently invoking replicated objects. The CMI protocol uses several high-level group communication services provided by JGroups to implement a number of features, including a distributed version of a JNDI-based resource registry, and a state propagation mechanism. More specifically, the distributed registry uses the *RPCDispatcher* and *StateTransfer* services of JGroups to guarantee that any changes made to registry by one of the servers are reliably propagated to the other servers (for instance, when a new object is created). The state propagation mechanism, in turn,

[4]In our work, we have used JOnAS version 5.1.0-M5. JOnAS is available at `http://jonas.ow2.org`.

uses the *MessageDispatcher* service of JGroups to guarantee that, whenever the server responsible for a replicated object fails, at least one of the remaining servers in the cluster will will have all the necessary information to continue responding to any ongoing or future client request on behalf of the failed server.

In the next subsection we describe how we have replaced, in the JOnAS source code, all JGroups services used in the implementation of the CMI protocol with the corresponding generic services provided by G2CL.

4.2. Migration Process

Our migration process was concentrated on two JOnAS classes, namely *SynchronyzedDistributedTree* and *JGMessageManager*. These are the main classes involved in the implementation of the distributed registry and the state propagation mechanism of CMI, respectively.

In both classes our migration strategy consisted, essentially, of changing all lines of code (and, when necessary, their associated configuration files) responsible for initializing the target JGroup services (i.e., *RPCDispatcher*, *StateTransfer* and *MessageDispatcher*), in order to replace them with the necessary code to initialize the corresponding services of G2CL.

One notable exception was the need to implement a new message serialization mechanism for JOnAS. This was required because the original version of JOnAS uses the serialization mechanism provided by JGroups, while jGCS (and, consequently, G2CL) leaves the serialization process to be implemented by the application.

Finally, we also had to change the way JOnAS handles the identification of group members. In the original version of JOnAS, group members are identified by the *Address* class of JGroups. In the new version, based on G2CL, this class was replaced by the *SocketAddress* class, which is the class used to identify group members in jGCS.

It is interesting to note that, even though the JGroups services we have replaced are actually *used* in many other parts of the JOnAS source code, we did not have to change any of those parts. This was due to our decision to keep the same JGroups API when implementing the corresponding services in G2CL.

Table 1 quantifies our migration effort in terms of the number of JOnAS packages, classes and lines of code (LoC) that had to be modified as part of our G2CL migration strategy. From that table we can see that most of the changes were performed in the CMI module, where about 2.5% of its packages, 2.8% of its classes and 11% of its lines of code had to be modified. These numbers reflect the fact that CMI makes intensive use of JGrous in its implementation, as we have explained above. Even though many of the changes made to the CMI module were certainly non-trivial, we can still see these numbers in a positive light if we consider that nearly 98% of the packages and classes of that module (comprising about 90% of its lines of code) were left unchanged after the migration. The percentage of changes in the other modules was much smaller, as expected, varying between 0.06 and 0.8%. Overall, we only had to change about 1% of the total of lines in the JOnAS source code.

Table 1. JOnAS migration numbers

	# Packages		# Classes		# LoC	
Module	**Total**	**Changed (%)**	**Total**	**Changed (%)**	**Total**	**Changed (%)**
CMI	80	2 (2,50%)	216	6 (2,78%)	18.691	2.106 (11,28%)
OW2-UTIL	235	1 (0,42%)	596	5 (0,84%)	33.538	270 (0,80%)
JOnAS	396	1 (0,25%)	2133	1 (0,04%)	180.030	111 (0,06%)
All	711	4 (0,56%)	2945	12 (0,41%)	232.259	2.487 (1,01%)

The above numbers are indicative that the programming effort required by the G2CL migration process was relatively low compared to the full size of the JOnAS source code. They also reflect the fact that group communication, although crucial to the provisioning of some important services of JOnAS, is only used scarcely in its implementation.

4.3. Performance Impact Analysis

Despite the clear software engineering benefits that can be associated with the use of generic APIs, one cannot neglect the inevitable performance impact that those systems may impose on the services they generalize.

With this concern in mind, we have analyzed the potential overhead caused by G2CL on the performance of JOnAS. Our analysis compared the performance of the original version of JOnAS, based on JGroups, against that of the new version, based on G2CL, using three different jGCS plug-in configurations.

4.3.1. Method

Our analysis was carried out in local cluster environment, which was configured in a manner to emulate a typical JEE clustering scenario [Lodi et al. 2007]. This environment was composed of nine PCs connected through a dedicated 10/100 Mbps Fast Ethernet switch. Each PC had the following configuration: Intel Core 2 Duo processor; 2 GB RAM (DDR2); and Linux Debian (version 5.0) operating system.

Six PCs were used in the business layer, each one running a separate JOnAS instance with CMI and the HA service enabled, playing the role of replicated EJB containers. Two other PCs were used in the presentation layer, each one also running a separate JOnAS instance, but now playing the roles of both web containers and CMI clients. Finally, one PC was used to run the Apache server (version 2.2.11), which was responsible for balancing the load amongst the servers of the presentation layer (see Figure 1).

To compare the performance of the different JOnAS versions, we have developed a simple EJB application with a single SFSB. This SFSB implements the basic functionalities of a shopping cart in an e-commerce application, offering operations to insert, update and remove items from the shopping cart. For persistence, we used the PostgreSQL relational database system (version 8.3) [PostgreSQL 2009]. This EJB application was installed in all the six servers of the business layer, with its

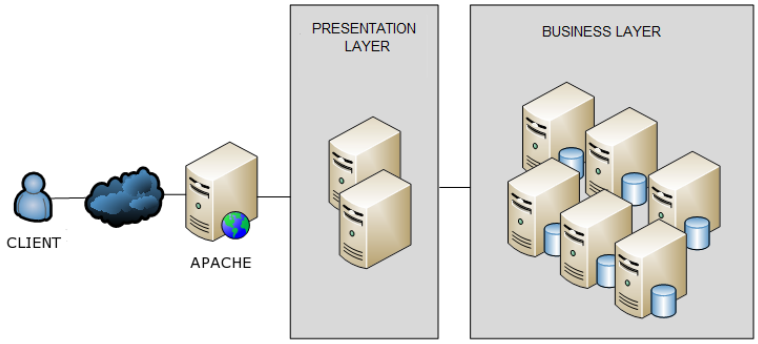

Figure 1. JOnAS evaluation environment

SFSB component being configured as a replicated CMI object.

We have also developed a simple web-based client application to continuously invoke a series of operations provided by the replicated object (shopping cart) at the business layer. Both the EJB application and the client application were implemented in a way to create an execution scenario similar to the one used by Lodi *et al.* in [Lodi et al. 2007], where the authors have compared the performance of an enhanced version of the JBoss application server [JBoss 2009].

We ran multiple sets of experiments, with each experiment involving a different version of JOnAS. In total, we analyzed the performance of four JOnAS versions: the original version, based on JGroups, and three variations of the new version, based on G2CL, using the jGCS plug-ins for JGroups, Spread and Appia, respectively. In all experiments we varied the number of clients from 50 to 100, so as to observe the performance of the different versions of JOnAS under different load conditions. To generate the client loads we used the ApacheBench(ab) benchmarking tool (version 2.0) [Apache 1996].

In terms of group communication features, we configured the three jGCS plug-ins to provide the same set of guarantees that is provided by JGroups in the original version of JOnAS. This was necessary to make sure that the new version of JOnAS, based on G2CL, would behave, at least functionally, in a similar fashion to its original version.

Finally, we used the *client response time* as our performance measure [Jain 1991]. In our analysis, this measured as computed by calculating the average response time observed across all clients during the same experiment. To achieve a confidence interval of 95%, each experiment was executed at least 30 times, with extreme outliers being removed using the *boxplots* method [Triola 1997].

4.3.2. Results

Figure 2 shows the average client response time observed for the four versions of JOnAS as a function of the number simultaneous client requests handled by the EJB application. As we can see, the different JOnAS versions are non-uniformly affected as the number of client requests grows. In addition, when we compare the

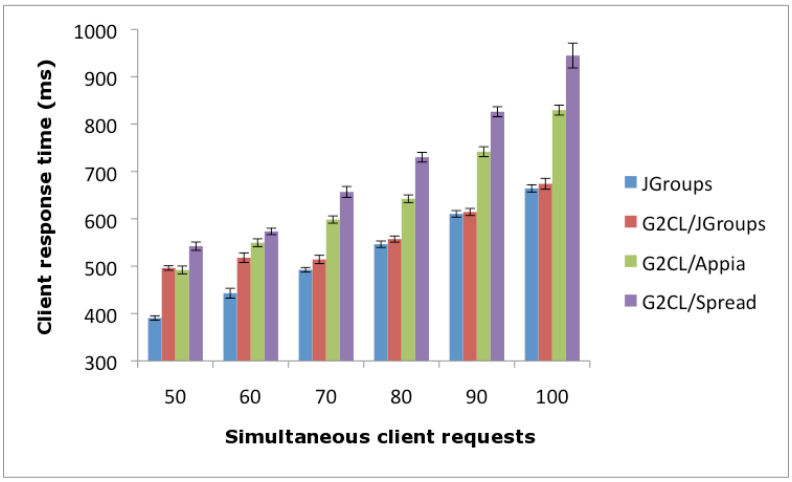

Figure 2. Performance analysis results.

original version of JOnAS, which uses JGroups directly, against the new version, based on G2CL configured with the JGroups plug-in, we note that their performances is very close, with a slight advantage to the former. This shows that the performance overhead imposed by G2CL on JOnAS is minimal (for 50 simultaneous requests, their performance differ by about 27% in favor of the original version, with that difference quickly falling below 5% as the number of simultaneous requests approaches the 70 mark).

We also observed that the new JOnAS version configured with the Appia plug-in imposes a virtually constant performance loss (in the order of 25%) when compared with its original version. When the new version is configured with the Spread plug-in, the observed performance loss is even higher (up to 42% for 100 simultaneous requests).

Even though none of the modified versions of JOnAS, based on G2CL, was able to beat the performance of the original version, we believe that the use of a generic API can still pay-off in terms of performance. As we have already shown elsewhere [Sales et al. 2009a, Sales et al. 2009b], Spread can outperform JGroups by a large margin under certain application scenarios, which means that for some distributed applications that use JGroups, the migration to a G2CL configuration based on Spread might actually result in a real performance gain. In this regard, we believe G2CL can offer a real contribution towards more effective clustering solutions, since it liberates developers to experiment with new group communication mechanisms without requiring a significant programming effort.

5. Conclusion

In this paper, we have presented our work on G2CL, a generic software layer providing a rich set of high-level group communication services. Our early experience in using G2CL in the the JOnAS application server as well as in other middleware technologies suggests that it can be effectively used as a generic group communication solution for existing clustered technologies, requiring a relatively modest migration effort and imposing a minimal performance overhead, particularly for those appli-

cations originally based on JGroups.

A natural line for future research is to improve G2CL with new group communication services and features. We are also conducting more case studies, involving open source clustered applications of varying sizes and domains, in order to better analyze its benefits and limitations. In particular, we plan to further investigate the conditions upon which migrating to G2CL would improve application performance.

Similarly to the other technologies it is based or relies upon, G2CL is being developed as an open source project. Its source code is freely available at `http://g2cl.googlecode.com`, under the *GNU General Public License* (version 2). We invite the interested readers to try it and also to contribute to its development.

References

[Alur et al. 2001] Alur, D., Malks, D., Crupi, J., Booch, G., and Fowler, M. (2001). *Core J2EE Patterns: Best Practices and Design Strategies*. Sun Microsystems, Inc., Mountain View, CA, USA, 2nd. edition.

[Amir et al. 2000] Amir, Y., Danilov, C., and Stanton, J. (2000). A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (FTCS-30, DCCA-8)*, pages 327–336, New York, NY, USA. IEEE CS Press.

[Apache 1996] Apache. Apache HTTP server benchmarking tool [online]. (1996). Available from: `http://httpd.apache.org/docs/2.0/programs/ab.html`.

[Ban 1998] Ban, B. (1998). Design and Implementation of a Reliable Group Communication Toolkit for Java. Technical report, Cornell University, Cornell University.

[Carvalho et al. 2006] Carvalho, N., Pereira, J., and Rodrigues, L. (2006). Towards a Generic Group Communication Service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA'06)*, pages 1485–1502, Montpellier, France. Springer.

[Chockler et al. 2001] Chockler, G. V., Keidar, I., and Vitenberg, R. (2001). Group Communication Specifications: A Comprehensive Study. *ACM Computing Surveys*, 33(4):427–469.

[Couloris et al. 2005] Couloris, G., Dollimore, J., and Kindberg, T. (2005). *Distributed Systems – Concepts and Design*. Addison-Wesley, Boston, MA, USA, 4th edition.

[Fowler 2004] Fowler, M. Inversion of Control – IoC [online]. (2004). Available from: `http://martinfowler.com/articles/injection.html`.

[Gamma et al. 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, MA, USA.

[Hedera 2008] Hedera. Hedera Group Communications Wrappers [online]. (2008). Available from: `http://hederagc.sourceforge.net/`.

[Jain 1991] Jain, R. (1991). *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, New York, NY, USA.

[JBoss 2009] JBoss. JBoss Application Server [online]. (2009). Available from: `http://www.jboss.org/jbossas/`.

[JGroups 2009] JGroups. JGroups – Building Blocks [online]. (2009). Available from: `http://www.jgroups.org/blocks.html`.

[JOnAS 2009] JOnAS. JOnAS – Java Open Application Server [online]. (2009). Available from: `http://jonas.ow2.org/`.

[JXTA 2008] JXTA. JXTA Community Project [online]. (2008). Available from: `https://jxta.dev.java.net/`.

[Lodi et al. 2007] Lodi, G., Panzieri, F., Rossi, D., and Turrini, E. (2007). SLA-Driven Clustering of QoS-Aware Application Servers. *IEEE Transactions on Software Engineering*, 33(3):186–197.

[Miranda et al. 2001] Miranda, H., Pinto, A., and Rodrigues, L. (2001). Appia – a Flexible Protocol Kernel Supporting Multiple Coordinated Channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS'01)*, pages 707–710, Phoenix (Mesa), Arizona, USA. IEEE CS Press.

[PostgreSQL 2009] PostgreSQL. PostgreSQL [online]. (2009). Available from: `http://www.postgresql.org/`.

[Sales et al. 2009a] Sales, L., Teófilo, H., D'Orleans, J., Mendonça, N. C., Barbosa, R., and Trinta, F. (2009a). An Evaluation of the Performance Impact of Generic APIs on Two Group Communication Systems. In *Anais do XXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC'09)*, pages 801–812, Recife, PE, Brasil. SBC.

[Sales et al. 2009b] Sales, L., Teófilo, H., D'Orleans, J., Mendonça, N. C., Barbosa, R., and Trinta, F. (2009b). Performance Impact Analysis of Two Generic Group Communication APIs. In *Proceedings of the 1st IEEE International Workshop on Middleware Engineering (ME'09)*, pages 148–153, Bellevue, WA, USA. IEEE CS Press.

[Shoal 2008] Shoal. Shoal – A Dynamic Clustering Framework [online]. (2008). Available from: `https://shoal.dev.java.net/`.

[SUN 2006] SUN. Java platform, enterprise edition (java ee) [online]. (2006). Available from: `http://java.sun.com/javaee/`.

[Triola 1997] Triola, M. F. (1997). *Elementary Statistics.* Addison-Wesley, Boston, MA, USA, 7th edition.