INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

JGroups evaluation in J2EE cluster environments

Takoua Abdellatif, Emmanuel Cecchet, Renaud Lachaize

N° 5336

Septembre 2004

THÈME COM

ISSN 0249-6399 ISRN INRIA/RR--5336--FR+ENG

JGroups evaluation in J2EE cluster environments

Takoua Abdellatif[*], Emmanuel Cecchet[†], Renaud Lachaize[‡]

Thème 1 – Réseaux et systèmes
Projet Sardes

Rapport de recherche n° 5336 – Septembre 2004 - 24 pages

Abstract: Clusters have become the de facto platform to scale J2EE application servers. Each tier of the server uses group communication to maintain consistency between replicated nodes. JGroups is the most commonly used Java middleware for group communications in J2EE open source implementations. No evaluation has been done yet to evaluate the scalability of this middleware and its impact on application server scalability.
We present an evaluation of JGroups performance and scalability in the context of clustered J2EE application servers. We evaluate the JGroups configuration used by popular software such as the Tomcat JSP server or JBoss J2EE server. We benchmark JGroups with different network technologies, protocol stacks and cluster sizes. We show, using the default protocol stack, that group communication performance using UDP/IP depends on the switch capability to handle multicast packets. With UDP, Fast Ethernet can give better results than Gigabit Ethernet.
We experiment with another configuration using TCP/IP and show that current J2EE application server clusters up to 16 nodes (the largest configuration we tested) can scale much better with this configuration. We attribute the superiority of TCP/IP based group communications over UDP/IP multicast to a better flow control management and a better usage of the network switches available in cluster environments. Finally, we discuss architectural improvements for a better modularity and resource usage of JGroups channels.

Keywords: J2EE cluster, group communication, JGroups, performance evaluation

[*] INRIA Rhône-Alpes and Bull – takoua.abdellatif@inrialpes.fr
[†] INRIA Rhône-Alpes – emmanuel.cecchet@inrialpes.fr
[‡] INRIA Rhône-Alpes and Institut National Polytechnique de Grenoble – renaud.lachaize@inrialpes.fr

Unité de recherche INRIA Rhône-Alpes
655, Avenue de l'Europe, 38330 Montbonnot-St-Martin (France)
Téléphone : +33 4 76 61 52 00 – Télécopie : +33 4 76 61 52 52

JGroups evaluation in J2EE cluster environments

Résumé: Les grappes sont actuellement la plate-forme standard pour construire des serveurs d'application J2EE qui passent à l'échelle. Chaque étage du serveur utilise des communications de groupe pour maintenir la cohérence entre les nœuds dupliqués. JGroups est l'intergiciel le plus souvent employé pour la communication de groupe dans les implémentations libres de serveurs d'applications J2EE. Mais à l'heure actuelle, aucune évaluation de ce logiciel n'est disponible, tout particulièrement concernant ses capacités de passage à l'échelle et de son impact sur les applications.
Nous présentons une évaluation des performances de JGroups dans le contexte des serveurs J2EE en grappe, avec différents réseaux d'interconnexion, différentes piles de protocoles et différentes tailles de grappes. Nous montrons qu'avec la pile de protocoles standard, les performances au-dessus d'UDP/IP dépendent beaucoup plus du commutateur utilisé que des performances brutes du réseau sous jacent. Un commutateur FastEthernet haut de gamme peut ainsi offrir de meilleures performances qu'un commutateur GigabitEthernet.
Notre second résultat est que TCP/IP offre, jusqu'à 16 nœuds (la limite de nos expérimentations) de bien meilleures performances qu'UDP/IP. Nous attribuons cette amélioration à un contrôle de flux optimisé ainsi qu'à un meilleur usage des commutateurs disponibles dans l'environnement des grappes. Finalement, nous discutons des améliorations possibles de l'architecture de JGroups pour avoir une meilleure modularité et une meilleure utilisation des ressources.

Mots clés: grappes J2EE, communication de groupe, JGroups, évaluation de performance

1 Introduction

As the popularity of dynamic-content Web sites increases rapidly, there is a need for maintainable, reliable and above all scalable platforms to host these sites. The Java™ 2 Platform Enterprise Edition (J2EE) specification addresses these issues. J2EE primarily targets four-tier application development [1]. The four tiers involved in a J2EE application server are: the Web tier (an HTTP server), the presentation tier (a web container providing Java Server Pages (JSP) or Servlet technologies), the business tier (an Enterprise Java Beans (EJB) container) and the database tier.

Figure 1 depicts a common J2EE cluster architecture. Clients issue HTTP requests from their web browser through the Internet. A number of front-end web servers handle the connections and forward the requests to one of the JSP servers according to a load balancing algorithm. JSP servers use group communication to replicate the state of each session so that if one server fails or becomes overloaded, the session can be handled by any other server. EJB servers contain the business logic and use group communication to replicate the naming registry (Java Naming Directory or JNDI) and replicate stateful session beans into memory. Scalability of J2EE clusters depends on (1) the ability of load balancing algorithms to evenly distribute the load among cluster nodes and (2) the scalability of group communication to limit the overhead of synchronization between replicated servers. In this report, we only focus on the latter problem tied to group communication scalability.

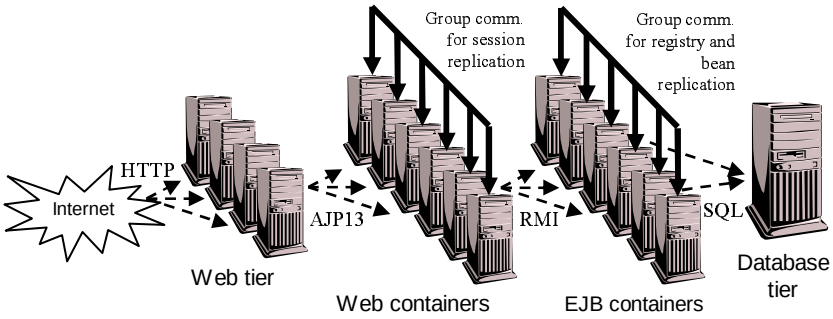

Figure 1. Overview of a J2EE cluster architecture

JGroups [2] (formerly JavaGroups) is the de facto group communication middleware used by open source J2EE application servers. However, no performance evaluation has been done regarding its scalability in this context. We are interested in studying the performance of JGroups in a cluster environment with different networking technologies (Fast Ethernet and Gigabit Ethernet), protocol stacks (UDP/IP multicast versus TCP/IP) and cluster sizes (from 2 to 16 nodes). For this purpose, we measure the raw throughput of JGroups using 1-n communications and we evaluate n-n communications that are representative of J2EE cluster group communications.

The outline of the rest of this paper is as follows. Section 2 presents the JGroups group communication middleware. Section 3 details how group communications are used in J2EE clusters and discusses related work. Section 4 presents our experimental methodology and results are discussed in section 5. Section 6 summarizes our evaluation and presents open issues. Section 7 concludes the paper.

2 JGroups

JGroups is an open source group communication middleware fully written in Java. Figure 2 gives an overview of JGroups' protocol stacks. The client (i.e. a J2EE application server instance) uses a high level abstraction called a *Channel*. A JGroups Channel can be considered as a group communication socket that the application uses to send/receive messages to/from the group. This abstraction makes the protocol stack transparent to applications. Therefore it is possible to reuse the same application code for different network configurations just by changing the JGroups protocol stack.

JGroups allows the programmer to build its own protocol stack using components that offer services such as reliability, ordering, fragmentation, group membership and so on. Reliability is implemented using negative acknowledgements (when a gap in the sequence numbers is detected). Missing packets are resent using unicast messages. Finally, the JGroups protocol stack interfaces with a network protocol such as UDP, TCP or any user defined protocol using a protocol adapter.

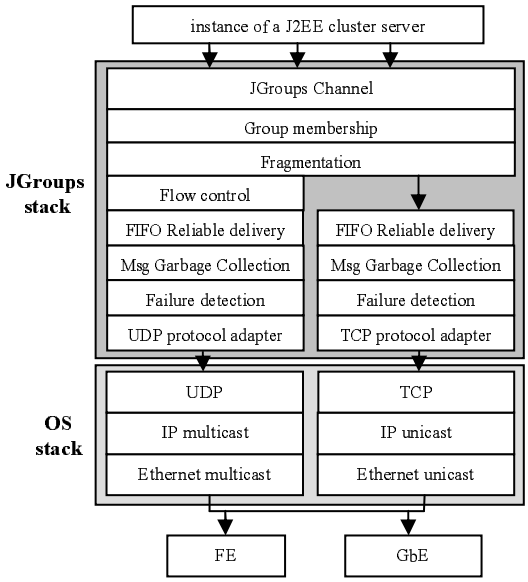

Figure 2. J2EE cluster group communication architecture with JGroups

2.1 JGroups with UDP and IP multicast

J2EE clusters running on clusters of workstations use FastEthernet (FE) or GigabitEthernet (GbE) networks with IP based communications. The cluster is usually equipped with a switch fabric that provides support for Ethernet multicast. Therefore it is possible to use JGroups with UDP and IP multicast to exploit at best the underlying hardware. However, as Ethernet is not reliable, it is necessary to include a component that ensures reliability and FIFO ordering in the JGroups protocol stack to ensure that all messages are delivered to all group members.

FIFO reliable delivery is implemented by the NAKACK and UNICAST layers in the JGroups stack. NAKACK has the responsibility to detect packet loss and to retransmit packets using the UNICAST layer for reliable point to point communication. Each member must keep a copy of messages multicast to the group for retransmission in the case of member failures or message loss. A message can only be garbage collected when all group members have received the message. This distributed garbage collection is implemented by JGroups' STABLE layer.

Additionally, JGroups offers two optional flow control layers. These layers reflect two kinds of flow control: reactive flow control and preventive flow control. With reactive flow control (like TCP/IP), the system detects a congested state and then throttles the senders by sending control messages. Reactive flow control (also called congestion control) is implemented by the FLOW_CONTROL layer in JGroups. With preventive flow control (also known as congestion avoidance), the system avoids congestion by conservatively limiting the outgoing message rate. This feature is implemented in JGroups by the FC credit-based protocol layer.

2.2 JGroups with TCP

TCP offers a point to point reliable communication with flow control. Group members are interconnected by point to point TCP connections and sending a message to the group consists in sending the message iteratively, within a loop, to each member on its dedicated connection. The NIO (New IO) architecture introduced in JDK 1.4 offers asynchronous IO to be performed on network connections. TCP-based multicast messages could benefit from this new implementation by parallelizing the TCP transfers.
Despite the reliability of TCP, there is still a need for reliability at the group level. Indeed, if the sender crashes before the end of the loop, the message will be received by just one part of the group members. JGroups discards messages that have not been received by all members to ensure multicast atomicity, i.e. that messages are delivered to all non-failed members or none.
As TCP already provides flow control, there is no need to use JGroups flow control layers. However, due to dependencies between STABLE and NACKAK layers, it is not possible to deactivate the distributed garbage collection (STABLE layer) which is useless in the TCP context. The same limitation forces the user to have reliable communications when the group membership service (GMS) is enabled. For example, reliability might not be necessary to multicast periodical monitoring information. Thus, the JGroups framework lacks flexibility in term of configuration since protocol layers have many dependencies between each other.

3 Group communication in J2EE clusters

In this section, we describe how group communications are used in the context of J2EE application server clusters. We present Tomcat in-memory HTTP session replication and JBoss clustering that are both using JGroups. Finally, we explain how the group communication patterns used in J2EE clusters can be simulated.

3.1 Tomcat in-memory session replication

Tomcat [10] is the servlet container used as the Java Servlet and Java Server Pages (JSP) technologies reference implementation. It is the most widely used web container in open source J2EE servers. The Java Servlet specification defines an HTTP session as a way to identify a user across more than one page request (i.e. visit to a Web site) and to store information about that user.
Tomcat provides the HTTPSession interface to create a session between an HTTP client and an HTTP server. The session persists in-memory for a specified time period, across more than one connection or page request from the user. The servlet programmer can bind objects to sessions, allowing user information to persist across multiple user connections. A typical session information is a shopping cart in an eCommerce application.
Tomcat clustering consists in running several instances of Tomcat on a cluster to achieve both fault tolerance and performance scalability. To prevent the failure of one node to cause the loss of the HTTP sessions stored on that node, HTTP sessions are systematically replicated among all Tomcat cluster nodes using JGroups. An HTTP session is always manipulated through *getter/setter* methods. The cluster implementation of the HTTP session multicasts every creation

and *set* call to the group to update the session state at each node. This way, if a node fails, the session state can be retrieved from any node in the cluster.
Tomcat version 4 uses the default JGroups protocol stack on UDP with a FIFO reliable delivery. All Tomcat cluster nodes belong to the same multicast group and use a single JGroups Channel to communicate all HTTP session updates. Note that update messages are all ordered (FIFO from sender) regardless of the session they update whereas ordering is only needed between updates of the same HTTP session.

3.2 JBoss clustering

JBoss is a popular open source J2EE server embedding Tomcat as its web container. Like Tomcat clustering, JBoss clustering [11] aims at improving scalability and high availability using replication techniques. JBoss relies on JGroups for the clustering of its naming registry (JNDI) and its EJB container.
Each JBoss instance has its own local JNDI. Each time an EJB is bound into a JNDI registry, the binding is propagated to all other registries of the cluster using JGroups. All JNDI instances maintain a coherent view of all beans bound in the JBoss cluster at any time, allowing any node to serve any registry lookup query.
For EJB container clustering, JBoss provides in-memory stateful session bean replication. The implementation is similar to the Tomcat in-memory session replication. All JBoss instances belong to the same group and multicast all session bean state changes through a single JGroups Channel. JBoss 3.x does not provide a distributed cache for entity beans but relies on database synchronization instead.
JBoss also uses the default JGroups UDP stack with an additional JGroups protocol layer called STATE_TRANSFER that allows a new member joining a group to initialize its state with the current state of other members. This is only used when adding nodes to a running system, so that they can catch up with the current cluster state and start serving requests as soon as they have joined the group.

3.3 Simulating group communications of J2EE clusters

A J2EE cluster processes a large number of concurrent queries issued by different users from various locations. If the processing power scales linearly by adding nodes in a cluster, performance can only scale linearly on read-only workloads where no group communication is needed for state updates between nodes. Every server instance in the cluster sends a multicast message to the other cluster members (including itself) each time a user interaction modifies the state of a replicated object in the system.

An efficient load balancing algorithm distributes the read and write requests evenly among cluster nodes. Therefore, each node processes the same number of multicast messages. The message size can vary from few bytes for replicating a simple identifier stored as an integer up to several megabytes for applications dealing with documents in the form of binary large objects.

On the one hand, the worst distribution scenario is like a master/slave model where a single node (the master) performs all updates and multicast them to all other nodes (slaves). In this case, we have *1-n group communications* (one to n) where *n* is the number of nodes in the cluster which is equivalent to the group size. On the other hand, an evenly distributed scenario features *n-n group communications* where all nodes multicast concurrently updates to all other nodes in the cluster. Any other load distribution can be represented by a m-n group communication pattern (m senders, n receivers) where $m \leq n$.

We can simulate the usage of group communications made by J2EE servers in a simple way. First, we create a single group between all cluster nodes with a FIFO reliable delivery property. As group membership is usually quite stable in clusters, we are not interested in measuring the cost of joining or leaving a group. Once all nodes have joined the group, a sin-

gle thread at each node can send asynchronous messages to the group to emulate session or state updates. To prevent any additional processing all received messages are simply discarded. We do not store the message content in memory to simulate object replication as this would interfere with the measurement of JGroups memory footprint.

3.4 Related Work

The JGroups middleware is widely used but only few micro-benchmarks comparing its performance against other group communication systems (GCS) have been published [4]. A more recent study [14] compares JGroups with Spread [16] and Appia [18] using 1-n and total order configuration. This comparison aims to evaluate the overhead of Java programming on GCS toolkits performance, compared to C++ implementations. Furthermore, this evaluation is done in the context of a specific scenario different from the J2EE replication model. We aim to benchmark the performance of JGroups for the simple "primary-backup" replication scheme used by most J2EE clustering implementations with different network setups. We do not compare different GCS implementations, but we rather focus on JGroups, the currently used GCS in J2EE clusters. In the same context, we tried to evaluate a Java RMI-based GCS called JGroup [3]. JGroup, presents memory limitations even for a few number of cluster machines and we could not get relevant results from this experience. In J2EE environments, another work [5] studies the pertinence of JGroups to build scalable Web services but does not evaluate its performance.

Previous performance evaluations already exist on other GCS at LAN scale. For example, an interesting study on different flow control mechanisms was already held for Isis [17]. However, in this study, Isis was always based on UDP or IP multicast. The use of TCP was considered prohibitive. In our work, we show that this is not always true and that the use of TCP flow control can give better results than two flow control mechanisms implemented on top of UDP in JGroups. We do not aim to discuss in this paper all flow control mechanisms but we just highlight that classical network protocols can improve group communication performance for free (without re-implementing new mechanisms at a higher level). Network technology has been improved in the last years, and multicast using TCP in a fully switched cluster can be more efficient than IP multicast. However, we cannot claim that this result is applicable to any other GCS. The GCS are so complex that we cannot predict the efficiency of an algorithm before testing it. For example, using a token ring as a flow control mechanism was efficient in Spread for the stability and delivery of messages. The same mechanism was tested in Isis and was not as interesting as in Spread [17].

To the best of our knowledge, at LAN scale, the traditional group communication systems were generally evaluated using 10 or 100Mb Ethernet, UDP and IP multicast. Ours is the first work to evaluate the performance of a group communication middleware using TCP and GigabitEthernet. Furthermore, unlike most GCS evaluations, we consider the worst case (n-n communication scenario) where the network and switches are stressed.

Furthermore, traditional group communication middleware evaluation focuses on the algorithmic aspect improvement to get scalability [12]. Our work is not aimed at improving current group communication algorithms but for a given algorithm implementation (JGroups one), we rather try to evaluate the influence of the network properties in term of bandwidth, latency and flow control to improve scalability.

Current trends in group communication research [6, 8] aims at improving the scalability of group communication middleware for systems with a very large number of members using probabilistic algorithms. These works do not apply to our target environment (J2EE clusters) that requires strongly reliable communications. Furthermore, these solutions are applied for groups of thousands of members dispersed on LAN networks with very dynamic membership properties. In our cluster environment, our scalability target does not exceed few hundreds of stable group members interconnected with a high speed LAN network.

Regarding architectural considerations, most GCS developed in the past 15 years share the same common features despite the differences between them [13]. There are some efforts that aim to get a configurable architecture. For example, Ensemble and JGroups offer a stack-based architecture. Spread distinguishes the transport part (in charge of the stability, delivery and ordering) from the network part (in charge of packet dissemination and reliability). This allows for example to get the network configuration transparent to the application's logic. However, this does not fulfill many configuration needs and new levels of abstractions have been proposed recently in [13]. This work argues in favor of less complex and more configurable group communication middleware. Our evaluation identifies similar issues showing that JGroups' abstractions are not well suited to J2EE clustered applications, the stack-based architecture is very costly and the stack protocol reconfiguration is not always possible (see section 6).

4 JGroups evaluation methodology

We evaluate the performance of JGroups in the presence of 1-n and n-n group communications. We vary the network technology (FE and GbE), the JGroups protocol stacks (UDP and TCP) and the cluster size (from 2 to 16 nodes). We describe hereafter the two benchmarks we have implemented, our experimental environment and finally our experimental methodology.
For both scenarios, we use asynchronous communications, i.e. all messages are sent in a burst without waiting for an acknowledgement (at the application level). An acknowledgement is only used at the end of a burst, to determine the elapsed time. Thus, these benchmarks allow to determine the upper bound performance that can be expected from JGroups with a given stack.

4.1 1-n group communication

This benchmark measures one-to-n group communication performance. One sender multicasts messages to all group members including itself. All members know, prior to the start of the experiment, the number of messages that will be sent. We have observed that we obtain stable results with 10,000 messages.
At the beginning of the experiment, the sender takes a local timestamp and starts sending 10,000 messages of a given size to the group. When a receiving node receives the last message, it sends an acknowledgement to the sender using a unicast message. The sender records the time taken by each node to receive all messages. We keep the time of the last acknowledgement, i.e. the time to multicast all messages for each experience. We ensure that the acknowledgements latency is negligible compared to the overall experience time.
This benchmark gives an upper bound of the group communication performance since it does not stress the network (no collision).

4.2 n-n group communication

This benchmark emulates n-to-n group communications. All nodes know *a priori* the number of messages they expect from other members (each sender sends the same number of messages). A barrier is used to synchronize the experiment start-up. Each node takes a local timestamp and starts sending its messages. When the last expected message from a sender is received, an acknowledgement is sent back in a unicast message to the sender. This allows stopping the timer at each sender.
Then, each sender calculates the time between the first multicast message sent and the acknowledgement message received by the last node receiving all sender's messages. For each sender, we calculate the average time of a multicast that is the time between a start-up message and the acknowledgement of the last message. Again, we ensure that the acknowledgements latency is negligible compared to the overall experience time.

This benchmark reflects a more realistic J2EE cluster scenario where server instances perform concurrent updates in the cluster. Both network and group communication layers are highly stressed by this test.

4.3 Experimental environment

We use HP rx2600 servers with the following components: dual 900 MHz Itanium-2 processors, 3GB DDRAM and a Gigabit Ethernet Broadcom BCM701 adapter. We use switched networks for all the experiments. FE results are obtained using the GbE adapters connected to a FE switch. Table 1 shows the raw performance of the various network technologies available in the HP rx2600. Latency and bandwidth over IP are measured with Netperf [9] between two machines.

Technology	Protocol	Latency (µs)	Bandwidth (Mb/s)
Fast Ethernet	UDP	40	94
Fast Ethernet	TCP	87	94
Gigabit Ethernet	UDP	32	835
Gigabit Ethernet	TCP	75	923

Table 1. Raw network performance on an Itanium-2 HP rx2600 machine

For each network, bandwidth is close to theoretical bandwidth. There is a factor of 9 between FE bandwidth and GbE bandwidth. Latency on top of IP is 30% higher for FE. These values are taken using a 3Com 3c16465A switch for FE and a HP Procurve 2724 GbE switch. We also use a Transtec 8tp GbE switch to study the switch impact on performance. Note that we do not use Jumbo frames with GbE. By varying the network technology and protocol stack in our experiments, we evaluate the impact of network latency and bandwidth on the group communication performance.

For all experiments, we use the Sun JVM 1.4.2_01 for Linux-IA64 with the following options: `-server` to use the server JVM instead of the client JVM, `-Xms768m` to set the initial Java heap size to 768 MB to prevent spending time in increasing heap size during application warm-up, `-Xmx768m` to set the maximum Java heap size to 768 MB, a typical J2EE server setting.

All machines run JGroups 2.2 and Linux Red Hat Advanced Server 2.1 with a 2.4.18-e.31smp kernel.

4.4 Experimental methodology

We use 4 different JGroups stacks for our tests, all of them providing the same delivery guarantees to the application (reliability and FIFO ordering). We present the protocol stack with the following template : *protocol_layer1(paramName=value, ..):protocol_layer2:..*

We present hereafter the 4 protocol stack configurations used in our experiments. The *bindAddress* parameter represents the address of the cluster node.

- *UDP*: default JGroups configuration on top of UDP/IP multicast as used by Tomcat and JBoss.

```
UDP(bind_addr=bindAddress;
    mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):
PING(timeout=3000;num_initial_members=6):
MERGE2:
FD(timeout=5000):
VERIFY_SUSPECT(timeout=1500):
pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
pbcast.STABLE(desired_avg_gossip=10000):
UNICAST(timeout=5000):
FRAG:
```

```
pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;
           print_local_addr=false)
```

- *UDP-FLOW*: same as *UDP* with reactive flow control.

```
UDP(bind_addr=bindAddress;
    mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):
PING(timeout=3000;num_initial_members=6):
MERGE2:
FD(timeout=5000):
VERIFY_SUSPECT(timeout=1500):
pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
pbcast.STABLE(desired_avg_gossip=10000):
UNICAST(timeout=5000):
FLOW_CONTROL
pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;
           print_local_addr=false):
FRAG
```

- *UDP-FC*: same as *UDP* with preventive flow control.

```
UDP(bind_addr=bindAddress;
    mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):
PING(timeout=3000;num_initial_members=6):
MERGE2:
FD(timeout=5000):
VERIFY_SUSPECT(timeout=1500):
pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):
pbcast.STABLE(desired_avg_gossip=10000):
UNICAST(timeout=5000):
pbcast.GMS(join_timeout=5000;join_retry_timeout=2000;shun=false;
           print_local_addr=false):
FC(max_credits=200000;min_credits=5200;down_thread=false):
FRAG
```

- *TCP*: JGroups stack based on TCP.

```
TCP(bind_addr=bindAddress;loopback=true):
TCPPING(initial_hosts=firstServer[serverPort]):
MERGE2:
FD(timeout=5000):VERIFY_SUSPECT(timeout=1500):
FRAG:GMS(join_timeout=3000;join_retry_timeout=2000)
```

The *firstServer* parameter is the address of the first started machine in the cluster.

We do not consider node failures in our experiments, therefore no group membership changes occur during the test.

For each benchmark and JGroups configuration, we vary the network technology (FE and GbE), the group size (with 2, 4, 8 and 16 nodes), and the message size starting with 1 byte increasing by a factor of 10 up to 10 MB. Each experiment is repeated 6 times and we present the minimum, maximum and mean results.

	1B	10B	100B	1kB	10kB	100kB	1MB	10MB
1-n	10,000	10,000	10,000	10,000	10,000	1,000	100	10
n-n	10,000	10,000	1,000	1,000	1,000	100	10	1

Table 2. Maximum number of messages of a given size that can be sent by JGroups without 'out of memory' errors for 1-n and n-n group communications, up to 16 nodes and 10,000 messages.

The number of messages sent by each sender is 10,000 for all experiments except the configurations mentioned in table 2 where a limit (lower size) is defined. This limit represents the maximum number of messages (rounded to a power of ten) that can be successfully sent without getting an 'out of memory' exception thrown by JGroups. These errors happen when the group communication middleware is overwhelmed with the message flow in which case it spends its time allocating buffers for retransmission and leads to memory exhaustion.
For large message sizes, JGroups can be overwhelmed with very few messages which is a real concern for clusters that must provide high availability.

We have made our benchmarks and complete stack configurations freely available for download from our web site (http://jmob.objectweb.org) to allow others to reproduce the results.

5 Experimental results

We call *throughput* the number of multicast messages per second that are delivered reliably to all nodes. In all graphs, the X axis represents the message size in bytes. Each bar in the graphs represents the mean result with extreme values for a group size of 2, 4, 8 and 16 nodes from left to right. Note that we abbreviate kilobytes by kB and kilobits by kb. We call useful bandwidth, the amount of application-level data (in bytes) delivered per second. The useful bandwidth ignores headers/trailers added by the different layers of the protocol stack. For the n-n benchmark, we only present the results obtained with the worst sender.
First, we present our JGroups performance evaluation with UDP configurations for both 1-n and n-n group communications. Then, we present TCP configurations results for both benchmarks. Finally, we show the network and protocol impacts on performance. Note that for most graphs, the Y axis is log scale.

5.1 UDP Results

Figure 3 shows the useful bandwidth in bytes per second using the 1-n communication benchmark with the JGroups UDP configuration described in section 4.4. Note that the y axis is log scale. We observe that the group size has no significant impact up to 8 nodes. JGroups reaches its maximum bandwidth of 5MB/s (40Mb/s) starting with message sizes of 10kB. We attribute the slight degradation observed with group size increase for small messages to JGroups' unicast acknowledgements.

In terms of number of messages per second, as illustrated in figure 4, for message sizes between 1 byte and 1 kB, about 800 group messages per second are delivered for cluster sizes up to 8 nodes. The 16 node configuration gives a performance that is 20% lower in average for the same sizes. Starting with message sizes of 10kB, the performance decreases linearly down to 0.5 message per second for 10MB.

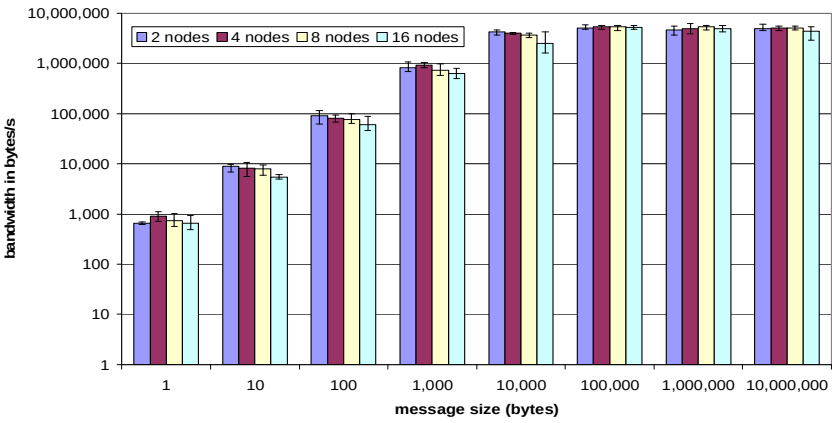

Figure 3. 1-n communication bandwidth in bytes/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

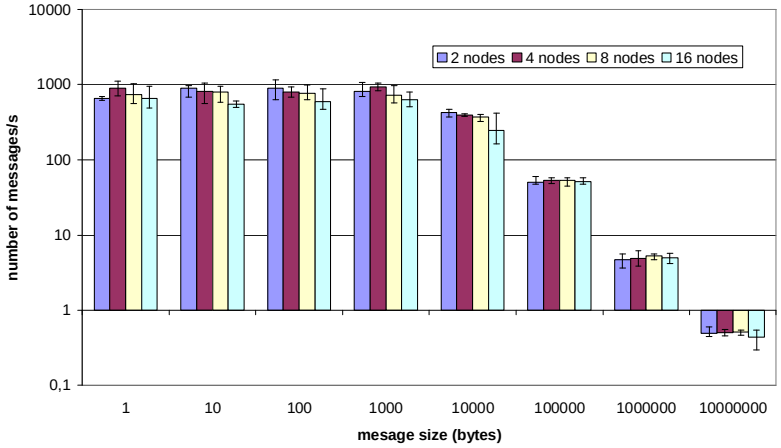

Figure 4. 1-n communication throughput in messages/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

The same experiments with the GbE network gives lower performance from 5 to 20% compared to FE. If the HP Procurve Gigabit switch limits the throughput degradation to 20% for all sizes, the Transtec Gigabit switch provides results more than 90% worse than the one obtained with FE. As we use exactly the same machines, network adapters and cables but just change the switch between experiments, this reflects the sensitivity of the group communication to the switch ability to efficiently handle Ethernet multicast packets. The switch vendors do not provide the accurate specifications regarding the processing of Ethernet multicast frames, making it very difficult to forecast the performance of UDP multicast.
Our experience shows that a high-end FE switch can give substantially better performance than entry-level GbE switches when using IP multicast for group communications.
Figure 5 shows the throughput per node in bytes per second using the n-n communication benchmark with the JGroups UDP configuration described in section 4.4. Figure 6 illustrates the same result in term of messages per second or throughput. The cluster size has a major impact on performance with a logarithmic degradation for message sizes up to 10kB. For a

message size of 1kB, the 2 node configuration is able to send more than 104 messages per second per node whereas the 16 node configuration only sends 1.2 messages per second.
When compared with the throughput of the 2 node setup, larger configurations show values dropping by an average factor of 2.6, 6.5 and 47.2 with 4, 8 and 16 nodes respectively. Extreme values are obtained for messages smaller than 1kB where a 16 node cluster can be up to 91 times slower than a 2 node cluster.
JGroups UDP offers a useful bandwidth of 1kB/s per node with 16 members. Throughput difference between 1-n and n-n benchmarks can be greater than a factor of 200. We attribute this degradation to the lack of flow control in the JGroups UDP configuration. The switch is overwhelmed with multicast packets that must be dropped. Similar results are obtained with both GbE switches. The indeterminism due to the switch behavior under saturation translates to a huge variance in the results.

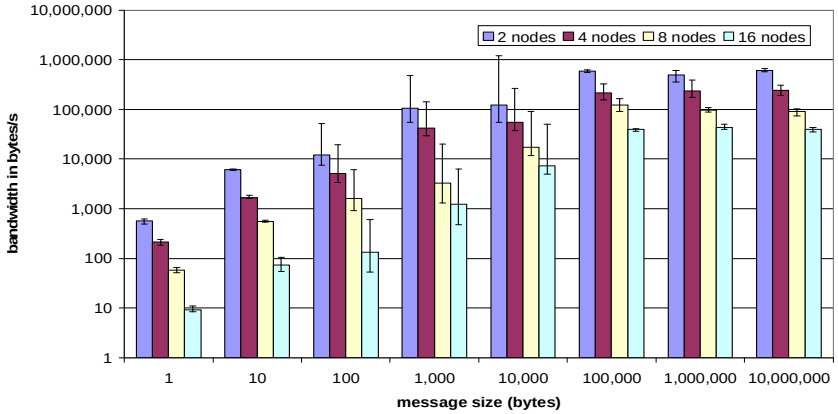

Figure 5. n-n communication bandwidth in bytes/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

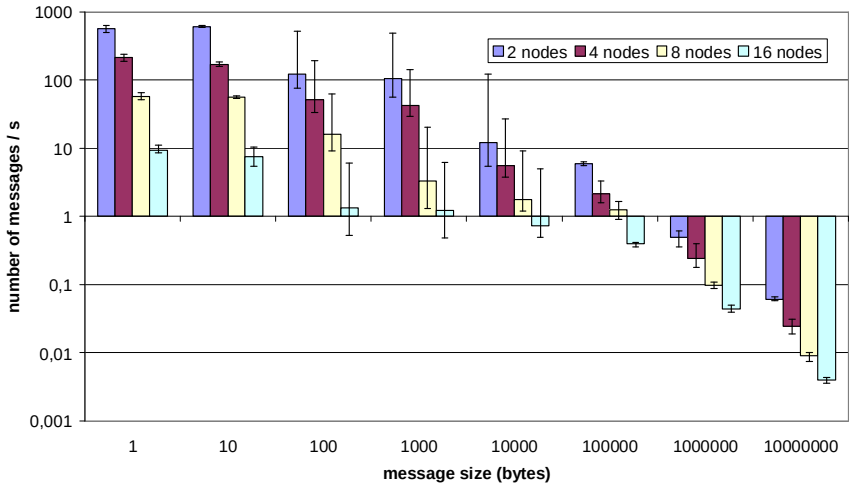

Figure 6. n-n communication throughput in messages/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

We have performed experiments with the *UDP-FLOW* and *UDP-FC* configurations described in section 4.4. None of these configurations was able to significantly alter the results we observed without flow control. Either the flow control is very conservative and the throughput is aligned on the slowest node (all nodes are running at the same speed but delays appear when messages are dropped), or flow control is very optimistic and the switch is flooded with multicast packets whereas the nodes do not detect overflow on their side. In both cases, the resulting performance does not exhibit any clear trend compared to the one observed without flow control. This result holds despite several efforts to tune flow control parameters. Figure 7 and figure 8 illustrate the percentage of throughput increase when using respectively *UDP-FC* and *UDP-FLOW* stacks with n-n communication on GbE.

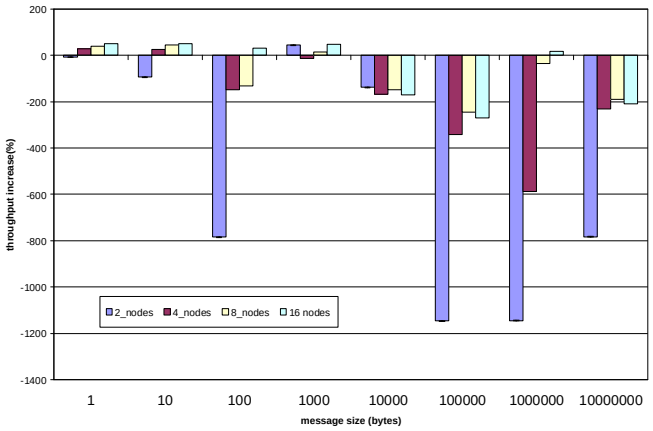

Figure 7. Percentage of throughput increase when using UDP-FC compared to UDP with n-n communication on GigabitEthernet

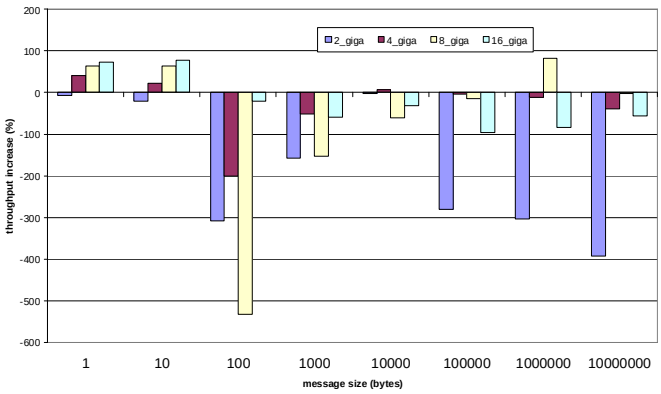

Figure 8. Percentage of throughput increase when using UDP-FLOW compared to UDP with n-n communication on GigabitEthernet

5.2 TCP results

Figure 9 shows the bandwidth using the 1-n communication benchmark with the JGroups TCP configuration described in section 4.4 and figure 10 shows the same results in term of throughput. Compared to UDP, results are much more stable with an insignificant variance in most cases.

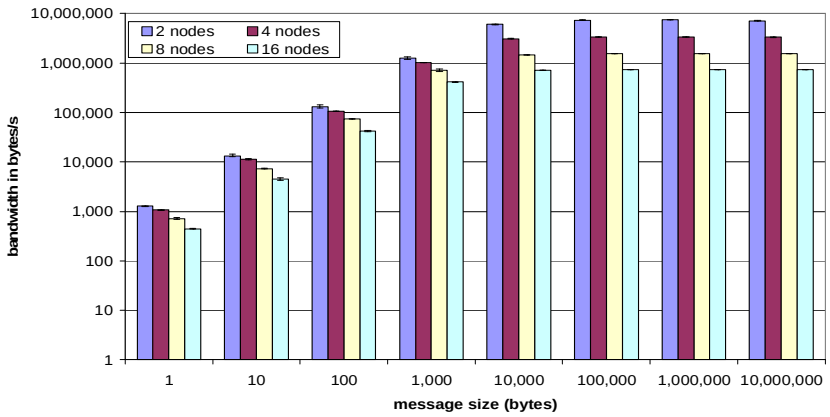

Figure 9. 1-n communication bandwidth in bytes/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

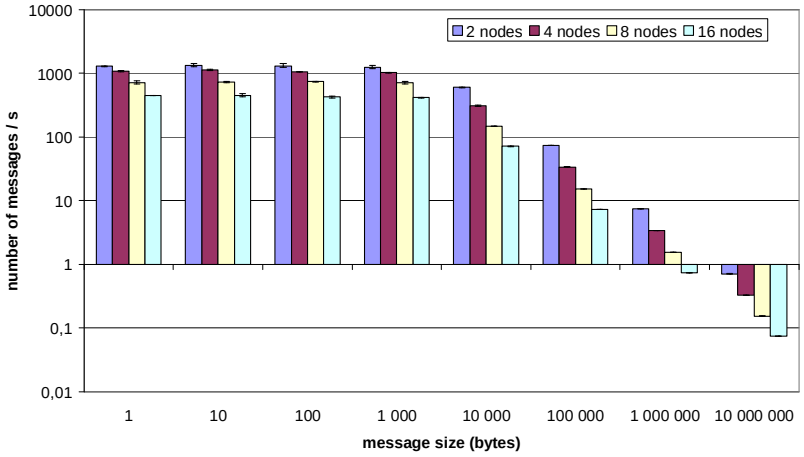

Figure 10. 1-n communication throughput in messages/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

	1-n communications			n-n communications		
Message size	4 nodes	8 nodes	16 nodes	4 nodes	8 nodes	16 nodes
≤ 1 kB	1.2	1.7	2.9	1.7	2.8	4.6
≥ 10 kB	2.1	4.5	9.5	1.8	3.2	6.1

Table 3. Throughput slowdown factor compared to a 2 node cluster for the 1-n and n-n group communications using the JGroups TCP configuration.

Group size influence on throughput varies according to message size as shown in table 3. Unlike UDP, we immediately observe the group size impact with the 1-n communication benchmark. As each message is sent on a dedicated connection for each group member, the

amount of data sent over the wire is multiplied by the group size compared to UDP/IP multicast where messages can be sent once to all members.
Throughput is stable between 1300 messages/s with 2 nodes and 450 messages/s with 16 nodes with message sizes ranging from 1 byte to 1kB. We notice a throughput decrease starting at 10kB. The corresponding useful bandwidth is peaking at 7.5MB/s (59Mb/s) with 2 nodes down to 740kB/s (5.9Mb/s) with 16 nodes for message sizes of 10kB and above. For a small group of 2 nodes, the useful bandwidth of JGroups with TCP is in average 50% better than with UDP. The peak bandwidth measured with GbE is 8.5MB/s (68Mb/s) with a similar impact of group size on performance.

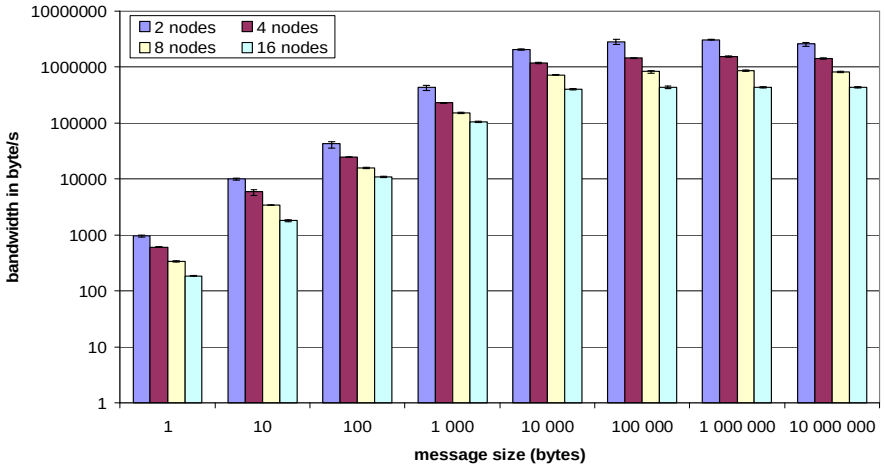

Figure 11. n-n communication bandwidth in bytes/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

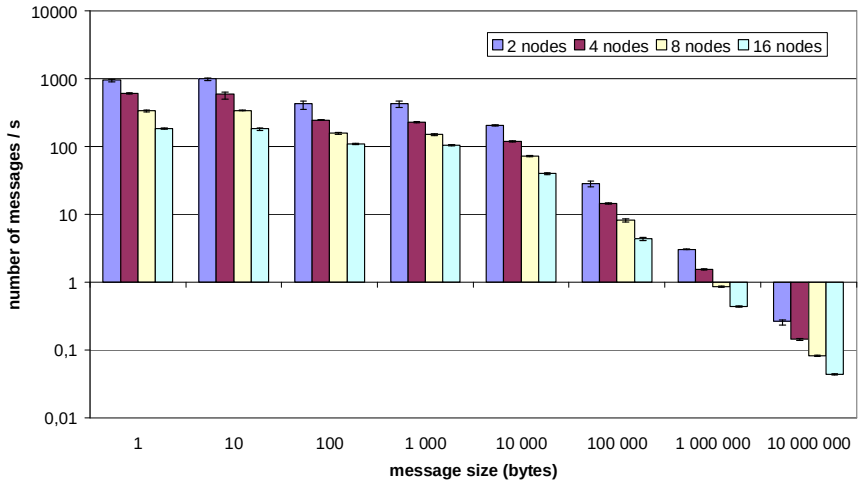

Figure 12. n-n communication throughput in messages/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

Figure 11 shows the throughput per node using the n-n communication benchmark with JGroups TCP configuration described in section 4.4 and figure 12 shows the throughput results. If the group size has still a significant impact on performance, the slowdown factor com-

pared to the 2 node configuration is different as reported in table 3. Even if a 16 node cluster is 8 times larger than a 2 node cluster, performance is only reduced by a factor of 4.6 for messages smaller than 1KB and by 6.1 with larger messages.
As a consequence, if we compare these numbers with those of TCP 1-n, the performance drop when going from 1-n to n-n communications affects more significantly large groups when messages are smaller than 1kB. For example, for a 1 byte message, a 2 node configuration has a throughput of 1200 msgs/s for 1-n communications and 966 msgs/s with n-n communications. A 16 node configuration for the same message size peaks at 447 and 185 msgs/s for 1-n and n-n communications, respectively. On the opposite, small groups proportionally observe a greater degradation than large groups on messages larger than 10kB. With a 10kB message size, a 2 node configuration delivers 601 and 205 msgs/s for 1-n and n-n communications respectively, whereas a 16 node cluster processes 71 and 40 msgs/s.
The maximum useful bandwidth is reached with message sizes of 10kB or more peaking at 3MB/s (24 Mb/s) for a 2 node cluster and at 437kB/s (3.5 Mb/s) with 16 nodes. GbE improves the useful bandwidth only for messages larger than 10kB and groups of 8 to 16 nodes, where the average increase is 11% and 36%, respectively.

5.3 Network impact on performance

As we already discussed in 5.1, the switch multicast capability is determinant on UDP/IP multicast performance. As no efficient flow control can be achieved with UDP configurations, the network bandwidth is of no interest and a GbE network provides worst results than FE in our tests.
Figure 13 shows the improvement of JGroups TCP 1-n communication throughput when using a GbE network instead of a FE network. Note that the Y axis is not log scale unlike previous graphs. We observe that there is no gain for messages up to 1kB. Cluster of 2 nodes can obtain a slight improvement of about 15% for message sizes of more than 100kB. For larger group sizes, the throughput increase for messages greater than 10kB is much more substantial starting at 55%, 104% and 139% up to 92%, 171% and 274% for clusters of 4, 8 and 16 nodes respectively. We explain this improvement by the fact that 1-n communications with TCP are exploiting the full network bandwidth and GbE provides better performance where FE saturates, that is to say for large group and message sizes.

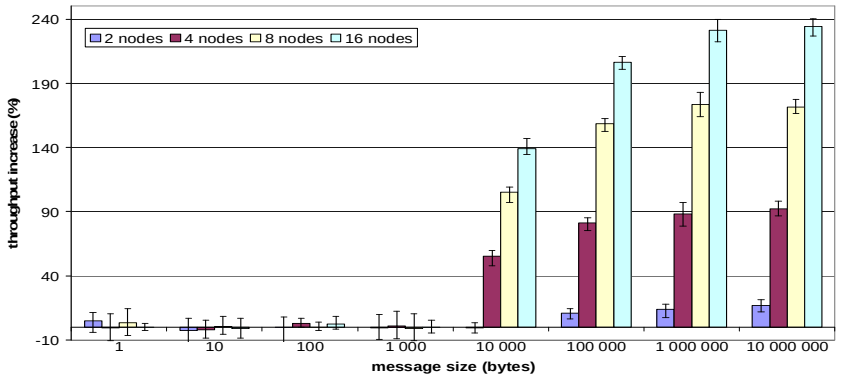

Figure 13. Percentage of throughput increase of GbE compared to FE using 1-n communications with JGroups TCP configuration.

In the case of n-n communications as represented in figure 14, there is no significant improvement for group sizes smaller than 8 nodes. The 8 node configuration benefits from about

10% throughput increase with GbE for messages of 100kB and above. The only configuration to get more than a 15% throughput improvement is the 16 node cluster where a 35% average increase is observed for messages greater than 100kB.

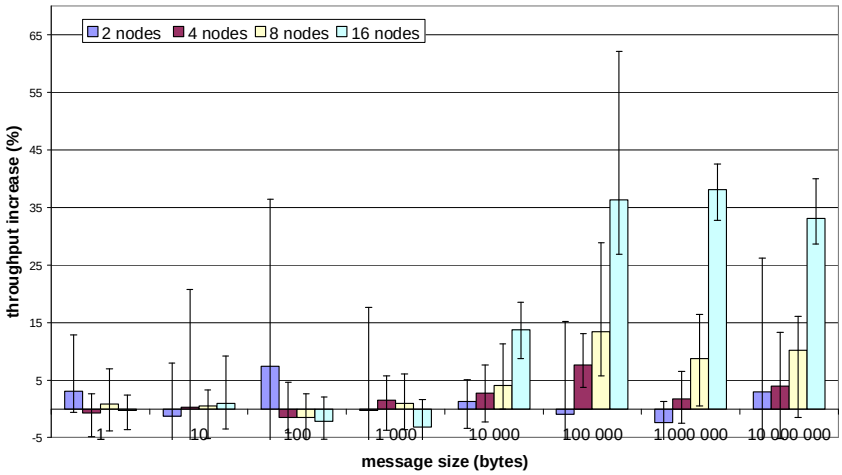

Figure 14. Percentage of throughput increase of GbE compared to FE using n-n communications with JGroups TCP configuration.

These modest improvements offered by GbE with n-n communications compared to 1-n communications is due to the group communication middleware. We have observed that, with 1-n communications, the performance is network bound whereas n-n communications are JGroups bound. Therefore, a J2EE cluster (up to 16 nodes) is more likely to not notice any performance improvement at the application level by replacing FE with GbE.

5.4 Protocol impact on performance

Figure 15 illustrates the improvement one can expect by using the TCP configuration we propose instead of the default UDP configuration described in section 4.4 for 1-n communications. Cluster of 2 nodes have a constant improvement of 50% in throughput except for 1 byte messages where it reaches 98%. Larger clusters perform better with UDP/IP multicast than TCP for messages greater than 10kB. In all cases, UDP is better than TCP for the 16 node cluster. In a 1-n communication, there is no flow control problem with IP multicast packets that performs much better than an iterative loop on TCP connections.
However, with n-n communications, the results are completely different as shown by figure 16. Note that Y axis is log scale in this figure. In all cases, the TCP stack outperforms the UDP results. For 10kB messages, the throughput increase with TCP compared to UDP ranges from 1500% with 2 nodes (2 MB/s versus 120 kB/s) up to 5300% with 16 nodes (400 kB/s versus 7 kB/s). The hugest performance gap is observed for 16 nodes and 1kB messages, with 8400 % more messages using the TCP configuration compared to the UDP configuration (104 kB/s versus 1.2 kB/s). As UDP results have a large variance, it happened in 2 measures (2 nodes with messages of 100 bytes and 1kB) that UDP and TCP performed comparably. But on an average of 6 runs for these 2 configurations, TCP obtains a throughput between 249% and 313% higher than UDP.

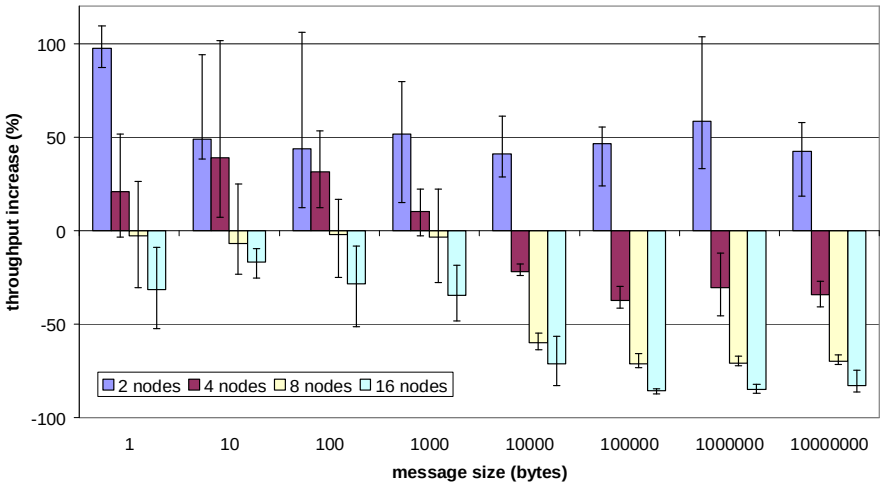

Figure 15. Percentage of throughput increase when using TCP compared to UDP with 1-n communications on Fast Ethernet.

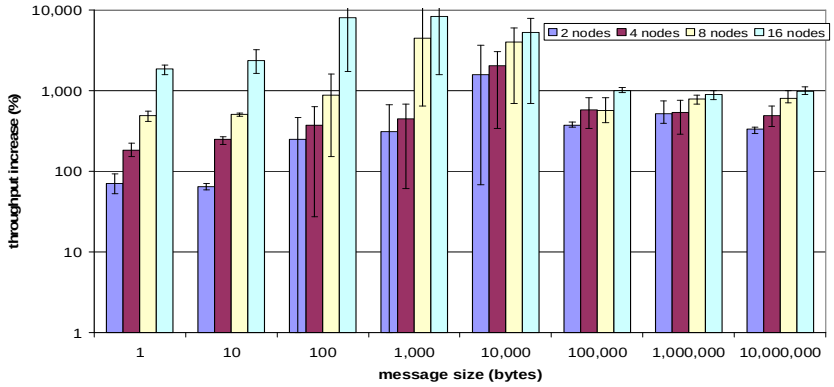

Figure 16. Percentage of throughput increase when using TCP compared to UDP with n-n communications on Fast Ethernet (y axis is log scale).

We attribute this superiority of TCP over UDP with n-n communications to the efficient flow control of TCP on point to point connections between group members. Network switches are also optimized to handle efficiently point to point connections that is best exploited by TCP connections than UDP/IP multicast. As an efficient UDP flow control would have to mimic TCP behavior by regulating the flow control on a per node basis, it is certainly more efficient to let the operating system TCP/IP stack do the flow control in the kernel rather than trying to emulate it in the group communication middleware.

6 Evaluation summary and open issues

Clustered J2EE applications are made of several multithreaded server instances, each thread of each instance handling a separate user session. This massive parallelism induces n-n group communications between server instances to replicate user sessions. The JGroups stack used in current implementations of popular open source application servers such as Tomcat or JBoss is

based on the UDP configuration we evaluated in this paper. We have shown that a 8 node cluster cannot send more than 3.3 group messages of 1kB per second with this configuration. This means that such a cluster cannot handle more than 26.4 updates per second in which case the group communication middleware becomes the bottleneck resource for the entire J2EE cluster.
A standard eCommerce workload as used in the TPC-W or RUBiS benchmarks [1] has 15% updates and 85% read-only queries. A server instance (a cluster node) is supposed to serve several hundreds or even thousands of user requests per second. If these 15% corresponds to 3.3 updates per second, it would mean a total throughput of 22 requests per second per server.
This lack of scalability can be addressed by relying on TCP for the transport layer. The exact same 8 node cluster with a JGroups stack based on TCP is able to handle 150.7 updates per second. If this represents 15% of the workload, each server instance would be able to handle more than 1004 requests per second.
In a master/slave scenario where only 1-n communications are performed, UDP scales better than TCP, but the performance is tightly bound to the switch optimizations for multicast packets. With a J2EE cluster workload where n-n communications are used, TCP scales better than UDP mainly due to the flow control at the connection level. Finally, in all n-n experiments, the bottleneck is the group communication middleware and therefore there is no significant gain by using GbE instead of FE. Besides, in a n-n scheme, the packet commuting capacity of the networking hardware is of uppermost importance compared to its bandwidth.
More complex network topologies (e.g. with a hierarchy of switches) should emphasize the observed results especially due to the inherent limitations of UDP (congestion prone) in such environments.
We think that it could be possible to improve performance further by having a protocol stack that takes better advantage of TCP. A JGroups protocol stack is configurable by adding needed services in a modular way, however there are many dependencies between these services. For example, it is not possible to have a group membership service (GMS) without distributed garbage collection (STABLE) and reliability based on acknowledgements (NAKAK). Some of these features are either inappropriate or redundant with those already provided by TCP (acknowledgements).
When multiple applications are running on the same Tomcat cluster, they fully share the same JGroups channel as shown on the left part of figure 17. This enforces an ordering of messages between all applications whereas ordering is only needed on a per application basis. If Tomcat nodes want to exchange some load information between each other, they are forced to use the ordered and reliable channel whereas none of these features might be requested.

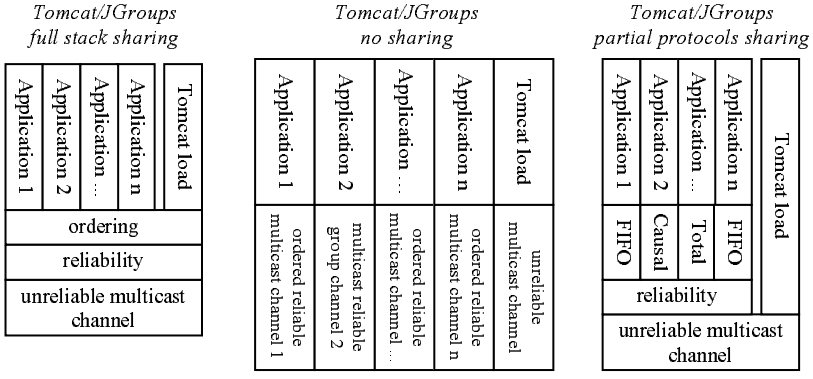

Figure 17. Possible protocols sharing in a JGroups stack for Tomcat

Another solution, as shown in the middle of figure 17, is to use a separate JGroups channel for each application but we have measured that, using the default configuration stack, a JGroups channel requires 28+x thread where x is the number of nodes in the cluster. A 8 node cluster

running n applications would require (28+8).n=36.n threads per node just for the group communication middleware. Given the high resource requirements for a group channel, users are forced to share a single channel for all their application needs as described in the previous case.
We think that there is a need for an intermediate solution where protocols are partially shared according to the requirements of the applications. The right part of figure 17 shows a configuration where the low level multicast channel (including group membership services) can be shared by all applications running in the Tomcat cluster. All applications requiring reliability features (potentially including view synchrony) could possibly share the same reliability layer but each of them has its own ordering module. The unreliable group channel required by Tomcat load information exchange can thus be plugged directly on the low level multicast channel. We are currently working on enhancing the modularity and configurability of JGroups protocols to provide JGroups' channels with a better resource usage.

7 Conclusion

We have presented an evaluation of JGroups, the most popular group communication middleware used in open source J2EE application servers. The raw throughput of JGroups, measured using 1-n communications, shows better UDP scalability compared to TCP for clusters larger than 4 nodes. UDP performance is directly related to the switch ability at handling multicast packets. In our experiments, FE always gives better results than GbE with UDP because our FE switch has a better processing of multicast packets than our 2 GbE switches. With TCP, GbE networks can at most double the performance for cluster of 16 nodes with messages larger than 1MB.
However, J2EE clusters are using n-n communications with all servers concurrently updating user sessions replicated cluster-wide. For all cluster and message sizes, the TCP stack improves UDP performance by a factor ranging from 1.8 up to 84. Even with various flow control settings for UDP, TCP constantly shows better throughput with a stable performance degradation when cluster size increases whereas UDP performance collapses with large clusters. Network bandwidth has no significant impact and GbE offers similar results as FE. Unlike 1-n communication that are network bound, n-n communications are JGroups bound.
An important result is that current J2EE application server clusters can improve their performance and scalability by replacing the default JGroups stack based on UDP with the configuration we proposed based on TCP. Besides, we think that further optimizations can be achieved by providing more flexibility in the configuration of the protocol stack.

Acknowledgments

We are grateful to Bela Ban, JGroups' developer, for answering our questions and helping us understanding JGroups' internals. We also thank Aurélien Dumez and Eric Ragusi, the administrators of the icluster2 at INRIA, for their support.

8 References

[1] Subrahmanyam Allamaraju, Karl Avedal, Richard Browett, Jason Diamond, John Griffin, Mac Holden, Andrew Hoskinson, Rod Johnson, Tracie Karsjens, Larry Kim, Andrew Longshaw, Tom Myers,Alexander Nakhimovsky, Daniel O'Connor, Sameer Tyagi, Geert Van Damme, Gordon van Huizen, Mark Wilcox, and Stefan Zeiger. – Professional Java Server Programming J2EE Edition – *Wrox Press*, ISBN 1-861004-65-6, 2000.
[2] JGroups : http://jgroups.org
[3] Hein Meling, Alberto Montresor, Ozalp Babaoglu, and Bjarne E. Helvik – Jgroup/ARM: A Distributed Object Group Platform with Autonomous Replication Management for Dependable Computing. Technical Report UBLCS 2002-12, October 2002
[4] JGroups Performance – http://www.jgroups.org/javagroupsnew/docs/performance.html
[5] Sing Li – High impact Web tier clustering, Part1. Scaling Web services and applications with JavaGroups – *Wrox press*, July 2003.

[6]Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky – Bimodal multicast – *ACM Transactions on Computer Systems 17*, May 1999.
[7] Raoul A.F. Bhoedjang, Kees Verstoep, Tim Ruhl, Henri E. Bal, and Rutger F. H. Hofman – Evaluation Design Alternatives for Reliable Communication on High-Speed Networks – *Proceedings of ASPLOS-9*, November 2000.
[8]Patrick Eugster, Sidath Handurukande, Rachid Guerraoui, Anne-Marie Kermarrec, and Petr Kouznetsov – Lightweight Probabilistic Broadcast –*Proceedings of DSN 2001*, July 2001.
[9] NetPerf – http://www.netperf.org/netperf/training/Netperf.html
[10] Jakarta Tomcat servlet container – http://jakarta.apache.org/tomcat/.
[11] Sacha Labourey and Bill Burke – JBoss clustering documentation – *JBoss Group LLC*, 2002.
[12] Ravi K. Budhia – Performance engineering of the Totem group communication system – *Distributed System Engineering 5*, pp. 78-87, 1998.
[13] Sergio Mena, André Schiper, Pawel Wojciechowski. – A Step towards a New Generation of Group Communication Systems – *Proceedings Middleware 2003*, Rio de Janeiro, Brazil, June 2003.
[14]Roberto Baldoni, Stefano Cimmino, Carlo Marchetti and Alessandro Termini – Performance Analysis of Java Group Toolkits : a Case Study –*Proceedings of FIDJI02*, pp. 81-90, 2002.
[15] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel – Performance Comparison of Middleware Architectures for Generating Dynamic Web Content – *Proceedings of Middleware 2003*, June 2003.
[16] Yair Amir and Jonathan Stanton - The Spread wide area group communication system - Technical Report CNDS 98-4, 1998.
[17] Kenneth P. Birman and Timothy Clark - Performance of the ISIS Distributed Computing Toolkit - Technical report TR 94-1432. Cornell University. June 1994.
[18] Appia Web Site - http://www.appia.di.fc.ul.pt.