

Evaluation of a group communication middleware for clustered J2EE application servers

Takoua Abdellatif, Emmanuel Cecchet, Renaud Lachaize

Sardes Project, LSR-Imag / Inria, 655, Avenue de l'Europe, 38330 Montbonnot, France
{first.last}@inrialpes.fr

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. 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

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 paper, we only focus on the latter problem tied to group communication scalability.

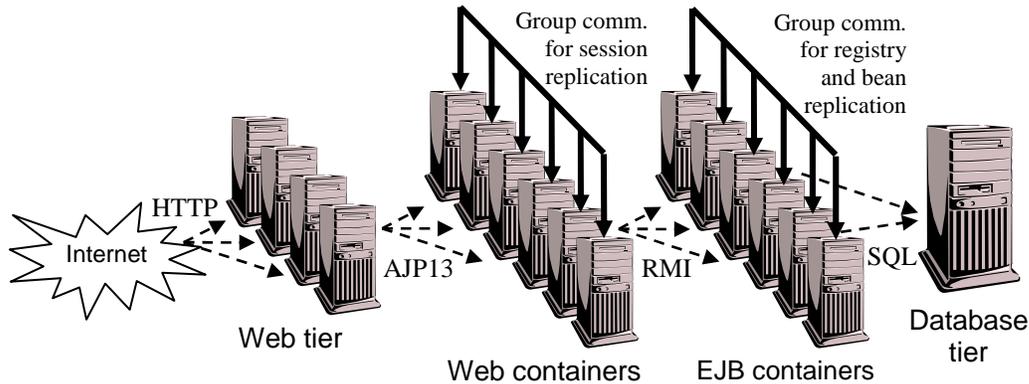


Figure 1. Overview of a J2EE cluster architecture

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. 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.

Ethernet multicast offers hardware support for IP multicast. This feature is exploited by JGroups when it is configured with UDP to multicast data to group members. We show that group communication using UDP/IP multicast are strongly correlated with the Ethernet switch multicast performance. In all our UDP experiments, our Fast Ethernet (FE) switch showed better performance than the Gigabit Ethernet (GbE) switches. Moreover, even with JGroups' UDP flow control modules, n-n communication throughput degrades quickly when message size increases.

JGroups can also be configured to use TCP/IP requiring a dedicated connection with each group member. We have employed a JGroups protocol stack using TCP/IP that provides the same user-level guarantees (ordering, reliability) as the UDP/IP multicast stack. We have measured that TCP n-n communications outperform UDP both in performance and scalability for all cluster sizes up to 16 nodes.

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 cluster of J2EE application servers 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.

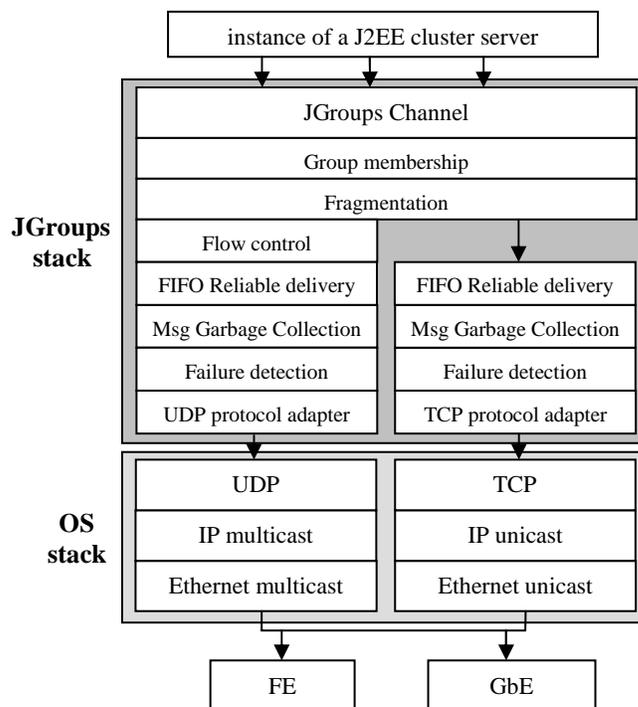


Figure 2. J2EE cluster group communication architecture with JGroups

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.

2.1 JGroups with UDP and IP multicast

J2EE clusters running on clusters of workstations use Ethernet networks (FE or GbE) 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. How-

ever, 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 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. Therefore, 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 server 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 the 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 single 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 the memory footprint of JGroups.

3.4 Related Work

The JGroups middleware is widely used but only few micro-benchmarks comparing its performance against other group communication middleware have been published [3]. A more recent study [14], compares JGroups totally ordered 1- n communications with other Java implementations in the context of replication in a three-tier architecture. Our study is not aimed at comparing JGroups with other group communication middleware, but to evaluate network technology and JGroups stack configuration impact on the performance of J2EE clusters. Other Java group communication middleware are based on RMI for the transport layer. We evaluated JGroup [2], an RMI-based group communication middleware but we were not able to obtain results due to runtime exceptions even for small clusters.

[4] studies the pertinence of JGroups to build scalable Web services but does not provide a performance study. To the best of our knowledge, ours is the first work to evaluate the performance of a group communication middleware in the context of J2EE clusters and its impact on the server scalability.

Current trends in group communication research [5, 8] aims at improving the scalability of group communication middleware for systems with a very large number of members like in peer-to-peer applications. These works do not apply to our target environment (clusters) that requires reliable communications based on high speed networks interconnecting a stable and bounded set of members.

Traditional group communication middleware evaluation focuses on the scalability of a specific algorithm such as totally ordered message delivery [12]. Our work is not aimed at improving current group communication algorithms but at evaluating various configurations and transport layers of an existing group communication middleware. We ensure that all configurations provide the exact same set of features to the application.

The majority of group communication systems developed in the past 15 years, share the same common features despite the differences between them [7]. New levels of abstractions have been proposed recently in [13]. This work argue 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.

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.

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 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 Itanium-2 node. Latency and bandwidth over IP are measured with Netperf [9] between two machines.

Technology	Protocol	Latency (μ s)	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 from JDK 1.4.2_01 for Linux-IA64 with the following options:

- `-server`: use the server JVM instead of the client JVM.
- `-Xms768m`: set the initial Java heap size to 768 MB to prevent spending time in increasing heap size during application warm-up.
- `-Xmx768m`: 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 reference these stacks as follows:

- *UDP*: default JGroups configuration on top of UDP/IP multicast as used by Tomcat and JBoss.
- *UDP-FLOW*: same as *UDP* with reactive flow control.
- *UDP-FC*: same as *UDP* with preventive flow control.
- *TCP*: JGroups stack based on TCP.

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.

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.

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 represent 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 the 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.

5.1 UDP Results

Figure 3 shows the throughput in number of messages delivered 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. 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.

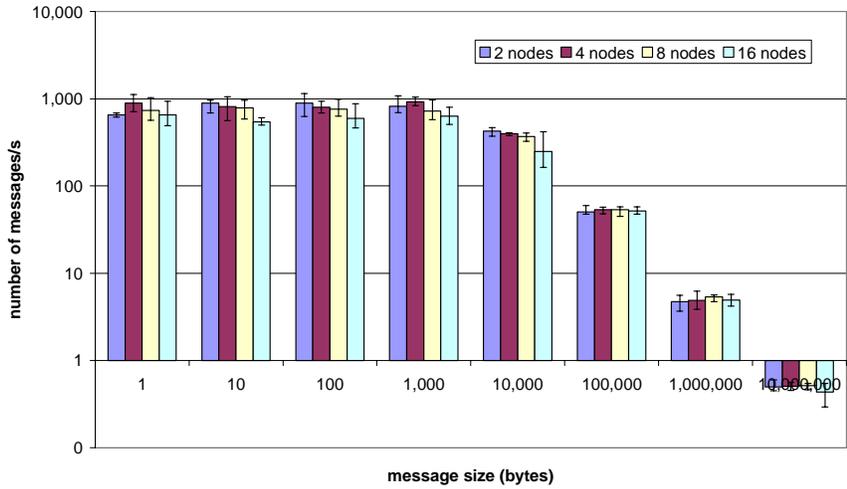


Figure 3. 1-n communication throughput in messages/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

Figure 4 presents the same results as figure 3 but in terms of useful bandwidth. It shows that 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.

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.

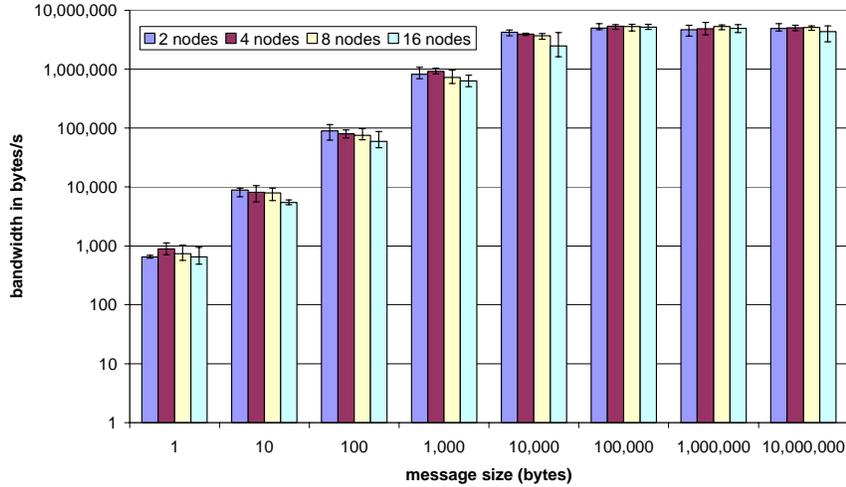


Figure 4. 1-n communication bandwidth in bytes/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

Figure 5 shows the throughput per node in number of messages delivered per second using the n-n communication benchmark with the JGroups UDP configuration described in section 4.4. 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.

If we observe the corresponding bandwidth presented in figure 6, JGroups UDP offers a useful bandwidth of 1kB/s per node with 16 members. Throughput difference between the 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 becomes 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.

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.

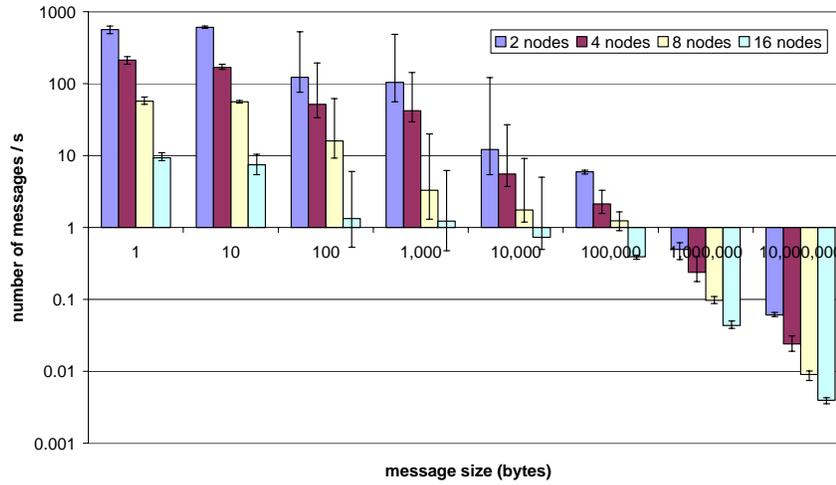


Figure 5. n-n communication throughput in messages/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

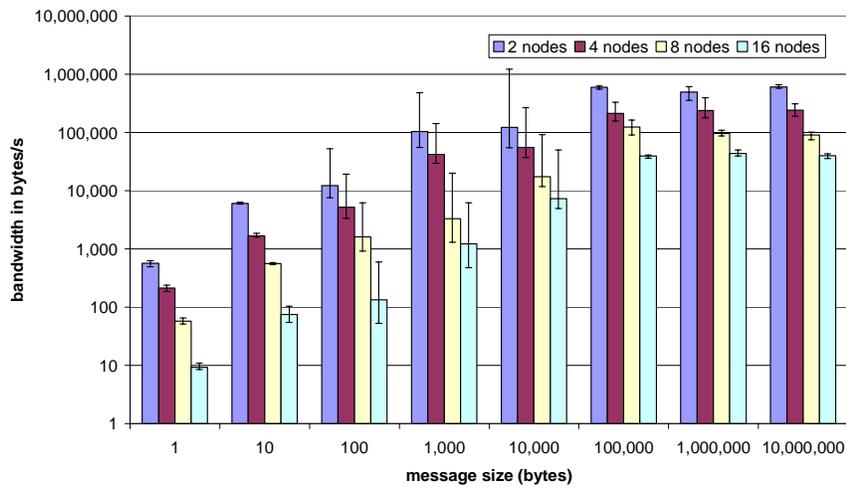


Figure 6. n-n communication bandwidth in bytes/s with JGroups UDP configuration on Fast Ethernet varying the message and group sizes.

5.2 TCP results

Figure 7 shows the throughput in number of messages delivered per second using the 1-n communication benchmark with the JGroups UDP configuration described in section 4.4. Compared to UDP, results are much more stable with an insignificant variance in most cases.

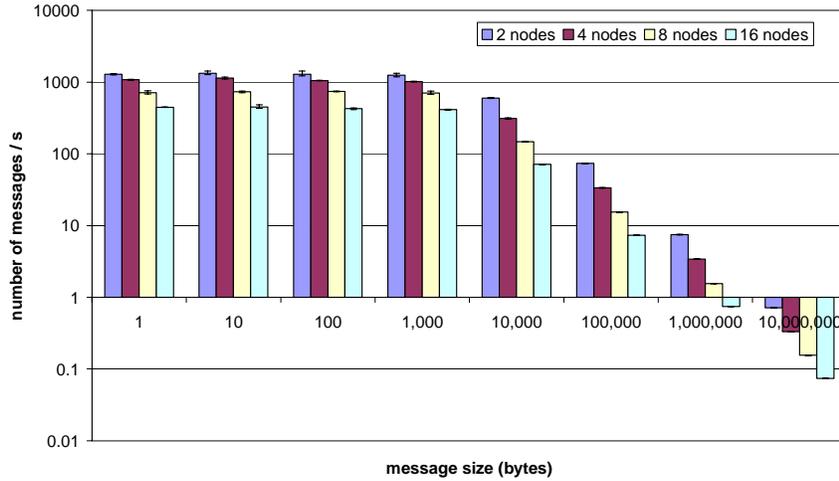


Figure 7. 1-n communication throughput in messages/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

The 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 the UDP/IP multicast case where the message can be sent at once to all members.

The 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. Figure 8 shows the corresponding useful bandwidth 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.

Message size	4 nodes	8 nodes	16 nodes
≤ 1 kB	1.2	1.7	2.9
≥ 10 kB	2.1	4.5	9.5

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

Figure 9 shows the throughput per node in number of messages delivered per second using the n-n communication benchmark with the JGroups TCP configuration described in section 4.4. If the group size has still a significant impact on performance, the slowdown factor compared to the 2 node configuration is different as reported in table 4. 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.

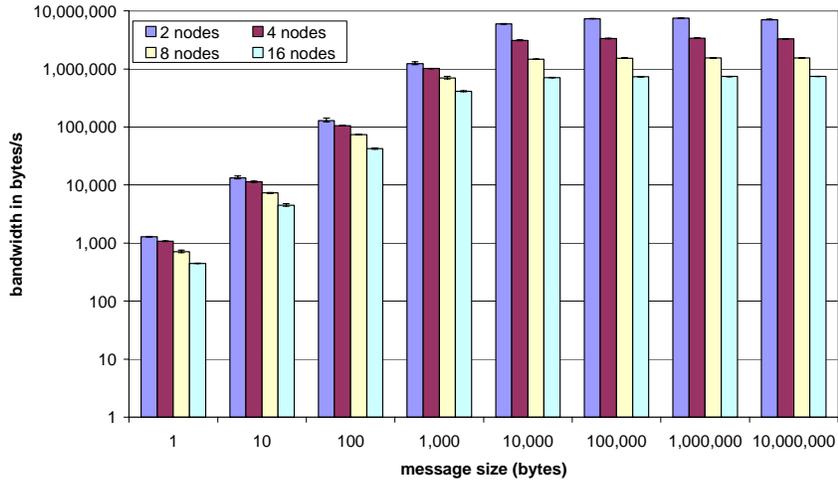


Figure 8. 1-n communication bandwidth in bytes/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

Message size	4 nodes	8 nodes	16 nodes
≤ 1 kB	1.7	2.8	4.6
≥ 10 kB	1.8	3.2	6.1

Table 4. Throughput slowdown factor compared to a 2 node cluster for the n-n group communications using the JGroups TCP configuration.

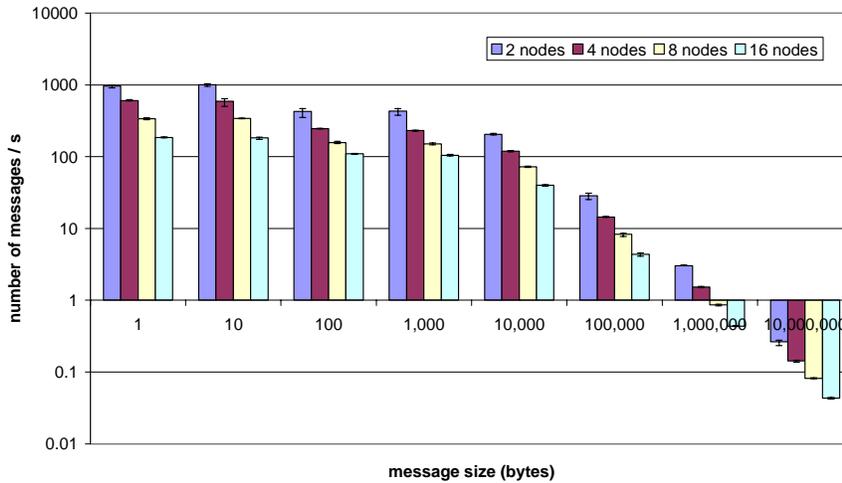


Figure 9. n-n communication throughput in messages/s with JGroups TCP configuration on Fast Ethernet varying the message and group sizes.

As a consequence, if we compare these numbers with those of table 3, 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 increase is in average 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 Gigabit network provides worst results than FE in our tests.

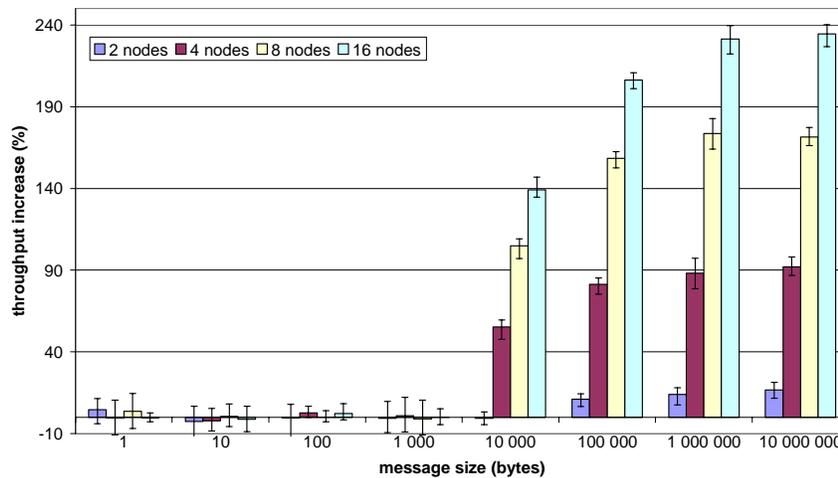


Figure 10. Percentage of throughput increase of GbE compared to FE using 1-n communications with JGroups TCP configuration.

Figure 10 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 3, 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 11 provides similar results but for n-n communications. There is no significant improvement for group sizes smaller than 8 nodes, and 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.

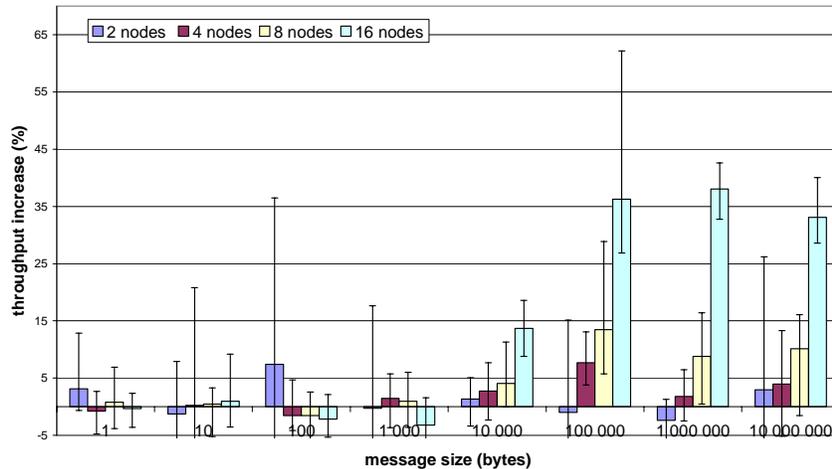


Figure 11. Percentage of throughput increase of GbE compared to FE using n-n communications with JGroups TCP configuration.

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 12 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.

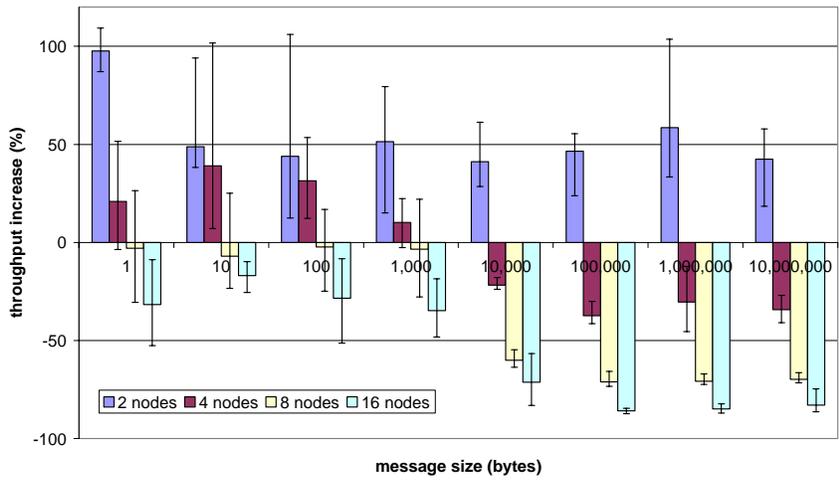


Figure 12. Percentage of throughput increase when using TCP compared to UDP with 1-n communications on Fast Ethernet.

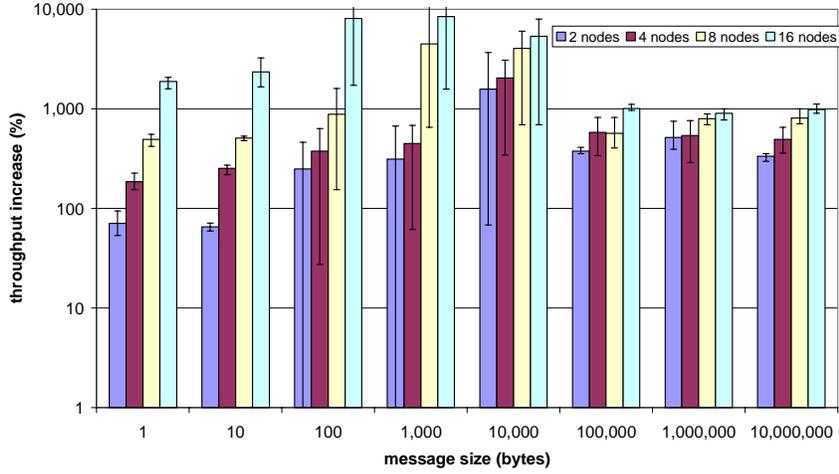


Figure 13. Percentage of throughput increase when using TCP compared to UDP with n-n communications on Fast Ethernet (y axis is log scale).

However, with n-n communications, the results are completely different as shown by figure 13. Note that the 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 up to 5300% with 16 nodes. The hugest performance gap is observed for 16 nodes and 1kB messages, with % more messages using the TCP configuration compared to the UDP configuration. 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.

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 (cf figure 5). 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 [15] 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.

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).

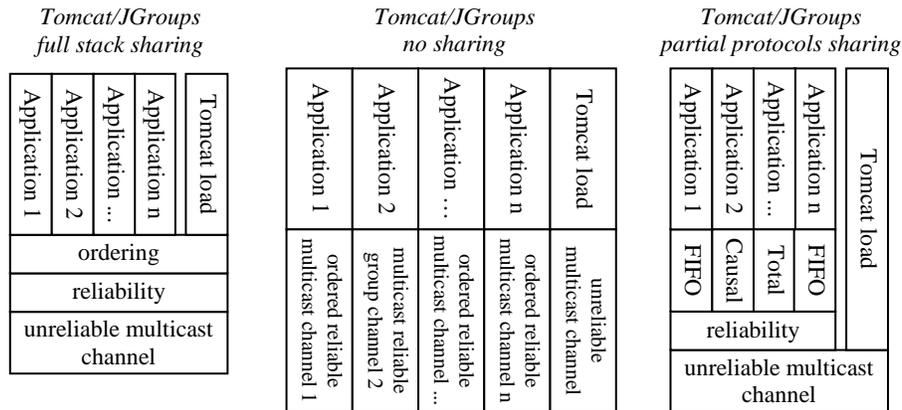


Figure 14. Possible protocols sharing in a JGroups stack for Tomcat

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 14. 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.

Another solution, as shown in the middle of figure 14, 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 14 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 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 on the results 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] S. Allamaraju et al. – Professional Java Server Programming J2EE Edition – *Wrox Press*, ISBN 1-861004-65-6, 2000.
- [2] B. Ban. – JavaGroups: Group Communication Patterns in Java – *Dept. Of Computer Science, Cornell University*, July 1998.
- [3] JGroups Performance – <http://www.jgroups.org/javagroupsnew/docs/performance.html>
- [4] S. Li – High impact Web tier clustering, Part1. Scaling Web services and applications with JavaGroups – *Wrox press*, July 2003.
- [5] K.Birman, M.Hayden, O.Ozkasap, Z. Xio, M. Budi, and Y.Minsky – Bimodal multicast – *ACM Transactions on Computer Systems* 17, May 1999.
- [6] R. A. F. Bhoedjang, K. Verstoep, T. Ruhl, H. E. Bal, and R. F. H. Hofman – Evaluation Design Alternatives for Reliable Communication on High-Speed Networks – *Proceedings of ASPLOS-9*, Cambridge, MA, November 2000.
- [7] G. V. Chockler, I. Keidar and R. Vitenberg – Group Communication Specifications: A Comprehensive Study – *ACM Computing Surveys*, vol.33, n°4, pp. 427-469, December 2001.
- [8] P. Th. Eugster, R.Guerraoui, S.B.Handurukande, A.-M.Kermarrec, P.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] S. Labourey and B. Burke – JBoss clustering documentation – *JBoss Group LLC*, 2003.
- [12] R.K. Budhia, L.E Moser and P.M Melliar-Smith – Performance engineering of the Totem group communication system – *Distributed System Engineering* 5, pp. 78-87, 1998.
- [13] Sergio Mena, André Schiper, and Pawel Wojciechowski – A Step towards a New Generation of Group Communication Systems – *Proceedings Middleware 2003*, Rio de Janeiro, Brazil, June 2003.
- [14] R. Baldoni, S. Cimmio, C. Marchetti, A. Termini – Performance Analysis of Java Group Toolkits : a Case Study – *Proceedings of FIDJI02*, pp. 81-90, Luxembourg, 2002.
- [15] E. Cecchet et al. – Performance Comparison of Middleware Architectures for Generating Dynamic Web Content – *Proceedings of Middleware 2003*, June 2003.