Evaluating Throughput Stability of Protocols for Distributed Middleware*

Nuno A. Carvalho José P. Oliveira José Pereira

Universidade do Minho

Abstract

Communication of large data volumes is a core functionality of distributed systems middleware, namely, for interconnecting components, for distributed computation and for fault tolerance. This common functionality is however achieved in different middleware platforms with various combinations of operating system and application level protocols, both standardized and ad hoc, and including implementations on managed runtime environments such as Java. In this paper, in contrast with most previous work that focus on performance, we point out that architectural and implementation decisions have an impact in throughput stability when the system is heavily loaded, precisely when such stability is most important. In detail, we present an experimental evaluation of several communication protocol components under stress conditions and conclude on the relative merits of several architectural options.

1 Introduction

Communication protocols used as components in distributed systems middleware range from the ubiquitous UDP/IP and TCP/IP Internet standards to custom protocols designed to address different reliability, ordering, performance, resource usage, and resilience requirements. In particular, multiparty or group communication protocols have been traditionally implemented at the application level and been highly relevant to middleware, for instance, to keep track of operational servers in a cluster and support load balancing of processing tasks across server clusters.

There has in fact been an increasing interest in group communication protocols such as JGroups [3], Spread [16] or Appia [10] in middleware supporting current multi-tier applications, towards both higher throughput and stricter consistency requirements. An example of this trend is the distributed software transactional memory proposed for FénixEDU [5]. Instead of relying solely on the underlying shared database management system to enforce consistency across different servers, updates are propagated and implicitly ordered using group communication. Another example is consistent database replication [8]. This allows concurrent conflicting updates to be processed by different replicas without fine synchronization thus enabling high performance. However, it ensures that all transactions are serialized and thus no conflicting updates are

*Partially funded by FCT through projects SFRH/ BDE/ 33304/ 2008 and PTDC/ EIA/ 72405/ 2006 (Pastramy), by PT Inovação S.A., and by ParadigmaXis S.A. The original publication is available at www.springerlink.com.

committed, avoiding the need for reconciliation or explicit sharing easing application development.

In fact, the motivation for this work was sparked by experimental observations when building and testing the ESCADA Replication Server,[1] a modular database replication protocol. Briefly, the testing setup used a cluster of servers, each running a PostgreSQL replica and ESCADA, with the workload of the TPC-C benchmark [17]. Under this scenario, one would observe that update dissemination would eventually slow down and the number of updates stored in memory grow. This was surprising, since the bandwidth being generated was easily achieved by a standalone benchmark of the group communication protocol in the same hardware setup.

This paper aims at explaining why the group communication within the larger application scenario would perform worse than in the standalone benchmark by testing the following two hypotheses:

- By running within a Java application with a large memory heap (i.e. the ESCADA Replication Server), the group communication protocol has to compete for memory with other threads, as the garbage collector represents an increasing share of the computation taking place.
- By running along a large number of interactive processes (i.e. instances of the PostgreSQL server) which together consume a substantial share of CPU bandwidth, the group communication has to compete for time slices.

Either way, the communication protocol would be unable to schedule events timely, for instance, to deal with window-based retransmission [6] implemented at the user level. This would prevent the protocol from fully exploiting available network bandwidth.

If true, this has an impact on architectural decisions when designing or selecting group communication protocols. Namely, a protocol made available as a library in Java should be particularly susceptible to the first. Any protocol that implements window-based mechanisms at the user level, regardless of using Java, is susceptible to the second. If true, this poses a challenge to using group communication in large servers running Java virtual machines with large heaps (e.g. application servers) or pools of interactive daemons (e.g. web or database servers).

Moreover, an in-depth knowledge of the dynamics of communication protocols in various workload conditions is also key to enabling self-managing distributed systems. In detail, being able to operate large and complex multi-tier applications depends on being able to ensure that individual system components are kept within their capacities to prevent congestion and trashing phenomena. If models underlying the creation of rule sets are unaware that communication capacity is degraded by server workloads, the resulting policies will be unable to keep the system within safe boundaries.

The rest of the paper is structured as follows. Section 2 we describe the communication protocols that we are evaluating and Section 3 we introduce our experimental setting. In Section 4 we present results that test each of the hypotheses. Finally, Section 5 discusses related work and Section 6 concludes the paper.

2 Protocols

To assess the stability of communication protocols for distributed middleware we select three kinds of protocols: point to point with network stack at kernel mode, such as

[1]http://escada.sf.net

Transmission Control Protocol (TCP) [6] or User Datagram Protocol (UDP) [14]; point to point with network stack at user mode, like LimeWire RUDP [9] or ENet [15]; and group communication protocols, such as Appia [10] or JGroups [3].

2.1 Point-to-Point in Kernel Mode

Simple point-to-point protocols implemented within the operating system kernel provide a baseline for comparison. First, the dynamics of TCP/IP in a number of environment conditions is well known and its implementation in mainstream operating systems is thoroughly tested and optimized. Second, because application level protocols are built on them, frequently on UDP/IP, and incur at least in the same overhead. Thus whenever possible, we test multiple APIs in C and Java, to discover also the impact of the Java Virtual Machine (JVM).

In detail, TCP protocol was assessed with three interfaces: the native BSD sockets interface in C, Java using `java.net` package and Java using `java.nio` package. Whenever possible, the same buffers are used for multiple I/O operations to reduce memory management overhead. In `java.nio`, direct byte buffers are used as the documentation describes them as improving performance.

The UDP protocol was evaluated in two implementations, C and Java using the `java.net` interface. Note however that UDP is not reliable and thus the amount of data sent differs from the amount of data received. This makes the tests useful only to determine baseline overhead.

Finally, the Stream Control Transmission Protocol (SCTP) [11] is aimed at combining the best features of TCP and UDP, ensuring the delivery of messages with or without order, has congestion control, allows the use of multiple streams and multihoming. These features can be switched on and off in contrast to the existing on TCP and UDP, and in this paper, a configuration similar to TCP has been selected.

2.2 Point-to-Point in User Mode

These protocols should provide an interesting indication of the cost of implementing reliability in user mode and in Java, when compared with point-to-point protocols in kernel. They should also provide an indication of the cost of group communication, when compared to such protocols.

The LimeWire application, implemented in Java, client of the Gnutella network, was the selected implementation for the evaluation of the Reliable User Datagram Protocol (RUDP), a lightweight version of TCP, whose features are: guaranteed delivery of messages, congestion control and retransmission of lost packets.

The ENet [15] protocol, reliable and in-order communication on top of UDP, was evaluated through their implementations in C and Java [18]. The evaluated versions were 1.1 and beta1, respectively.

2.3 Group Communication Protocols

Appia [10] is an open source layered communication toolkit implemented in Java providing extended configuration and programming possibilities. The Appia toolkit is composed by a core that is used to compose protocols and a set of protocols that provide group communication, ordering guaranties, atomic broadcast, among other properties. Appia is a protocol kernel that offers a clean and elegant way for the application to express inter-channel constraints. In assessing this toolkit only one process writes the

Resource	Properties
Processor	2 $\times$ 2.4 GHz AMD Opteron (64 bits)
RAM	4 GBytes
Operating System	Linux 2.6.12-16 (Ubuntu Kernel)
Network	Gigabit Ethernet

Table 1: Configuration used for benchmarking.

data and another reads it, creating a point-to-point channel in the group membership. The evaluated version was 4.1.0.

JGroups [3] is a group communication toolkit implemented in Java, which offers reliability and group membership on top of TCP or UDP. Its most powerful feature is its flexible protocol stack, which allows developers to adapt it to exactly match their application requirements and network characteristics. Like Appia, was selected for this evaluation to measure the impact of increased network stack, particularly being it in user space. Once again, in assessing this toolkit only one process writes the data and another reads it, creating a point-to-point channel in the group membership. The evaluated version was 2.6.3 GA.

Spread [16], another group communication toolkit, consists of a library that user applications are linked with, in this evaluation our application was implemented in Java, and a binary daemon which runs on each computer that is part of the processor group. In this combined implementation, which delegates the communication work to other process, we are particularly interested in observing the impact of the Garbage Collector in throughput stability. The evaluated version was 4.0.0.

3 Experimental Setting

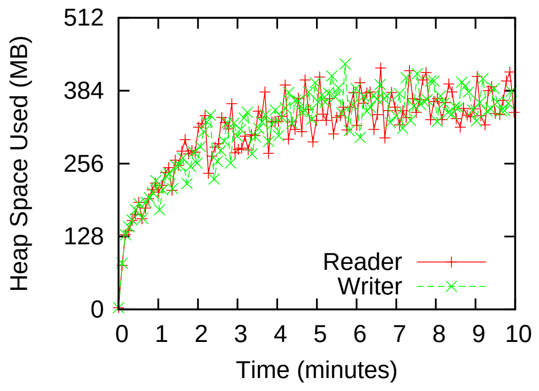
3.1 Hardware and Software

The experimental evaluation described in this paper was performed using two HP Proliant dual Opteron processor machines with the configuration outlined in Table 1. The operating system used is Linux, kernel version 2.6.22-16, from Ubuntu. The C programs are compiled with GCC 4.1.3 without any special flags. The Java based evaluations are compiled and run with Sun's Java 1.6.0_03. The availability of multiple CPU cores allows us to assess also the ability of protocols to take advantage of this features, which is increasingly important in current hardware configurations.

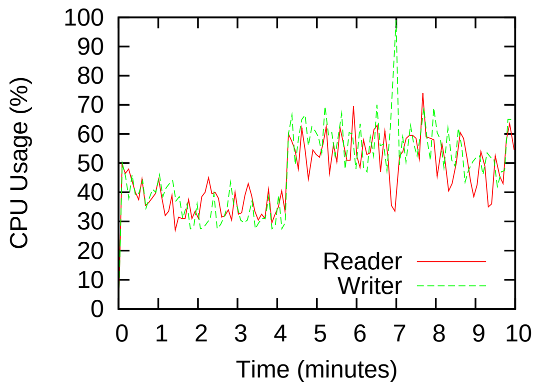
3.2 Measurements

A run consists in having one process sending messages as fast as allowed by the communication protocol (i.e. the *Writer process*), while in a different machine another process reads them also as fast as possible (i.e. the *Reader process*). No artificial delays are inserted in any of them. Although experiments have been reproduced with different sizes, this paper includes only results obtained by writing and reading data in 2000 bytes chunks. Test machines are otherwise idle, to avoid disturbing measurements.

Measurements are done concurrently by running Dstat [19] every second. Dstat is a standard resource statistics tool, which collects memory, disk I/O, network, and



(a) Java heap size



(b) CPU usage (100% means fully using 2 cores)

Figure 1: Resource usage with the Garbage Collector workload.

CPU usage information available from the operating system kernel. All measurements therefore include all load on each of the machines. These are saved to a log file and later processed off-line to extract the results presented in the paper.

Each run lasts 10 minutes. Communication starts after 4 minutes. Measurements during the first 4.5 minutes and the last 30 seconds of each run are discarded. This allows background workload generators to warm up and wind-down without impacting results. When fully automated, test runs described in this paper take approximately 10 hours to run and produce 15GBytes of log files.

3.3 Background Workload Generators

The first competing background workload generated aims at reproducing the conditions in a loaded server, in which a large number of processes or threads alternate between idle and busy periods and compete for CPU. Due to the common operating system scheduler policy of favoring interactive processes, this workload cannot be duplicated simply by having a single background process in an infinite loop. Instead, we use the operating system clock to determine at each time what share of the CPU has been used and have a pool of processes alternate between idle and busy periods to meet the desired

CPU occupancy. This strategy affects all processes, because the load is imposed on the scheduler, which is also affecting processes in kernel mode.

The second competing background workload generator aims at reproducing the conditions in a loaded Java based single virtual machine server, in which a large heap is being managed. Due to common garbage collector optimizations, it is not enough to simply allocate memory in tight loop, since every allocation is short lived and favors generational garbage collectors. Instead, we build random linked structures such that probabilistically some elements become unreferenced and others are added at the same rate.

In this paper we tune the parameters of this workload generator such that it uses approximately 384MBytes out of a maximum of 512MBytes allocated to the virtual machine, as can be observed in Fig. 1(a). The resulting usage of CPU is shown in Fig. 1(b), which shows the garbage collector workload alone up to minute four and then the cumulative effect of a test run with a TCP/IP socket. The target CPU occupancy of the CPU workload generator was then set at 60%, to allow direct comparison with the garbage collector workload generator. Note that this corresponds to slightly more that the load that one to the two cores can handle.

4 Results

4.1 Unstable Protocols

Unfortunately we were unable to make all target group communication protocols run the proposed test successfully. Namely, Spread daemons would disconnect either the sender or the receiver and we were unable to finish any test run. This behavior is well known and expected, having been thoroughly discussed in the supporting mailing list, since Spread does not do end-to-end flow control and expected the application to do it. We were also unable to reliably complete test runs with the Appia protocol, although it has end-to-end flow control implemented by “memory managers”. It would either block or crash with out-of-memory errors.

The same problem happened with some of the point-to-point protocols being used for comparison. Namely, both implementations of eNet would consume an ever increasing amount of memory at the sender, leading to trashing or an out-of-memory crash. The LimeWire RUDP protocol, although stable, would not be able to use a significant portion of the available bandwidth and thus does not provide an interesting comparison. We do however understand that this is most likely a design decision and not a bug, since the typical usage of Limewire RUDP will have multiple concurrent connections over a single residential network link (e.g. ADSL).

4.2 Scenario 1: No Competing Workload

Running all protocols without any competing background workload provides a baseline for later comparison as well as a first measurement of the resources required to saturate the 1GBits network. The results are shown in Fig. 2 and Fig. 3.

As shown in Fig. 2, bare TCP/IP and UDP/IP are always able to saturate the network, regardless of the API being used in C or Java. However, as seen in Fig. 3, there is a large CPU overhead when using Java even if we took care not to allocate memory for each operation, i.e. we always write from and read to the same buffers. Using the novel NIO interface with direct buffers does not make a noticeable impact.

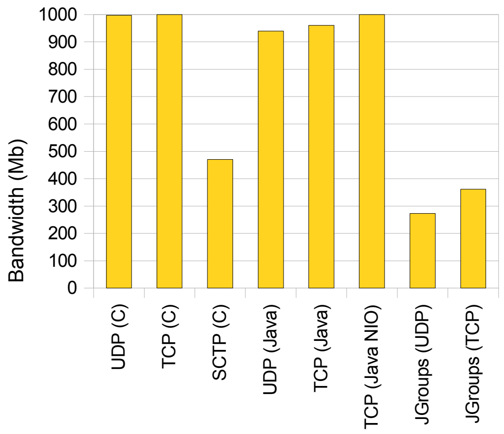
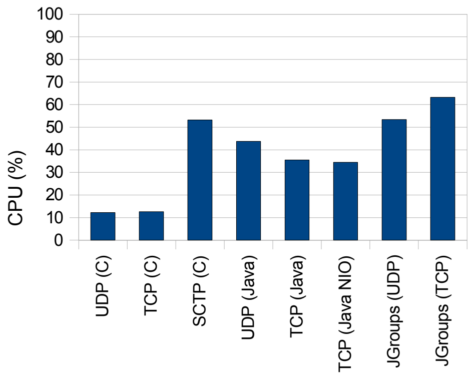
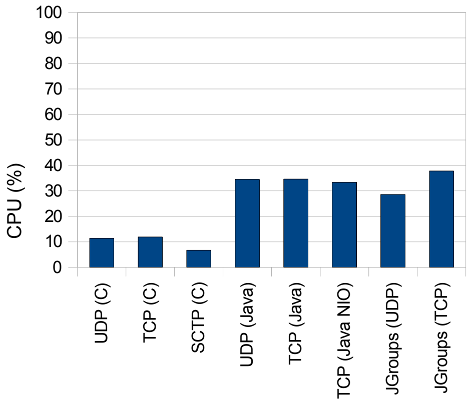


Figure 2: Bandwidth usage without background workloads.



(a) Measured at the Writer process.



(b) Measured at the Reader process.

Figure 3: CPU usage without background workload (100% represents the two CPUs of the machine).

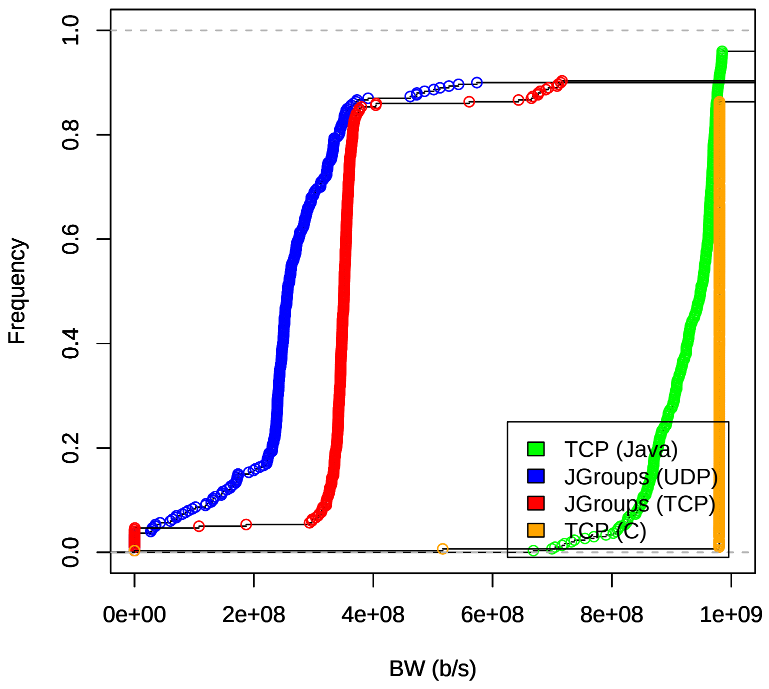


Figure 4: Distribution of bandwidth in time periods.

Regarding SCTP, also implemented in the operating system kernel, it is interesting to note that it does not fully saturate the network. The reason for this seems to be that the sender is fully using one of the available CPU cores. In contrast, CPU usage seems to be asymmetric, as the receiver is much less intensive.

Both configurations of JGroups tested are unable as well to saturate the network. Again, the reason seems to be that the sender fully uses one of the two available CPU cores and is unable to exploit the second. Interestingly, the TCP/IP configuration is able to achieve slightly higher throughput and use the second CPU core to some extent. This is probably true as TCP/IP processing is done within the kernel and thus scheduled to multiple cores.

Finally, besides average throughput, it is interesting to note how each option is able to sustain such throughput stably, without variation. Fig. 4 plots the empirical cumulative distribution function of bandwidth observed in each period of time. A straight vertical line or steep slope denote low variance while a moderate slope or staircase denote high variance. It can be observed that TCP in Java is more unstable than in C, and that the UDP configuration of JGroups more unstable than the TCP one.

Lessons Learned: These results point out that one should make as much use of kernel based TCP/IP as possible and avoid Java in the implementation of group communication. Otherwise, one should make the protocol multi-threaded and account for additional CPU usage.

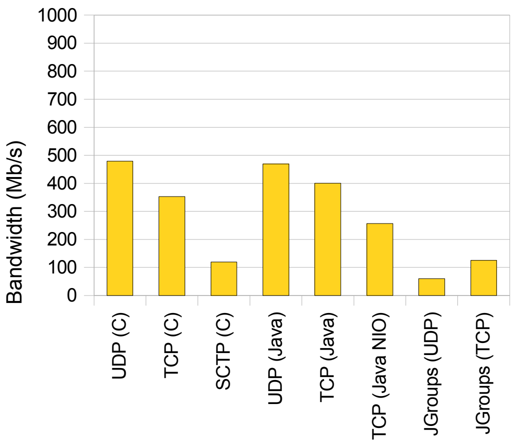


Figure 5: Bandwidth usage with competing CPU workload.

4.3 Scenario 2: Competing CPU Workload

As described in Section 3, this scenario adds competing background CPU workload, thus introducing scheduling latency in user level processes. The impact on average bandwidth is shown in Fig. 5, showing that all protocols have their throughput reduced.

Fig. 6 shows the same results as a fraction of the original maximum achieved with each protocol, making it easier to evaluate which protocols suffer the most. The least affected is UDP, although this is misleading since UDP is not reliable and is discarding some traffic. Most interestingly, SCTP is one of the protocols that is most affected, even if it is implemented in the kernel. Moreover, the Java NIO interface performs worse that the original Java sockets interface.

Regarding group communication protocols, the UDP configuration is much more affected than its TCP counterpart. This confirms our hypothesis that it is hard to have a retransmission algorithm in user mode when there is a competing workload and consequence scheduling latency.

Finally, Fig. 7 shows that throughput stability suffers with any of the protocols, which exhibit similar variability, confirming that there is no significant disadvantage of Java in this scenario.

It would also be interesting to perform the experiments using a real time scheduling class for protocol threads. This is however not straightforward for two reasons. The protocols that need it the most, such as JGroups, are implemented as libraries and this might require elevating the privileges of the Java virtual machine as a whole, which is undesirable. Second, since the protocol is itself responsible for a substantial share of CPU usage, it could seriously degrade the performance of the entire service.

Lessons Learned: These results reinforce that one should make as much use of kernel based TCP/IP as possible in the implementation of group communication. Otherwise, one should make the protocol multi-threaded and account for additional CPU usage.

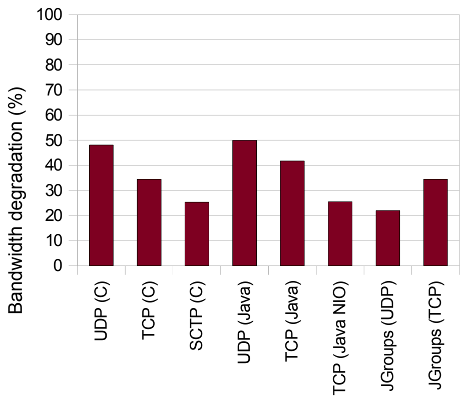


Figure 6: Bandwidth degradation with competing CPU workload.

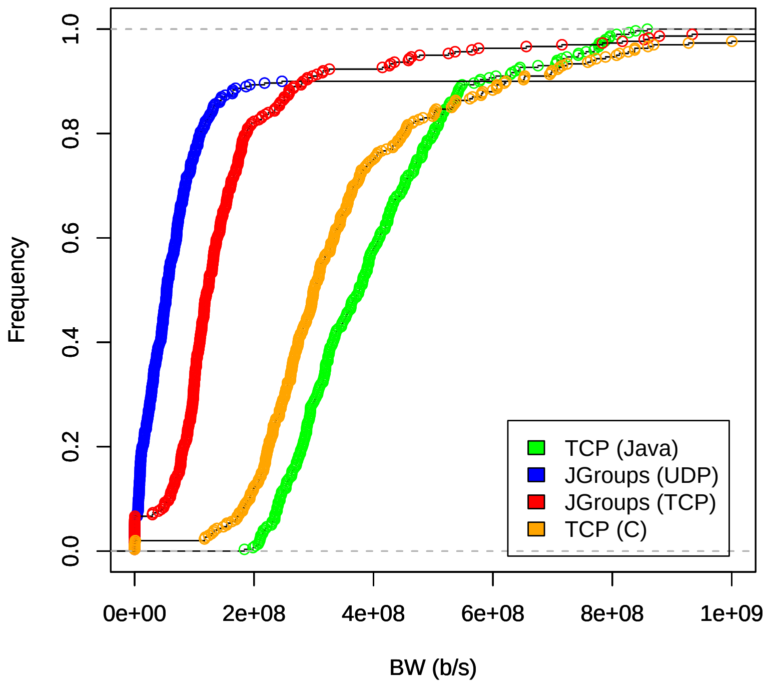


Figure 7: Distribution of bandwidth in time periods with CPU workload.

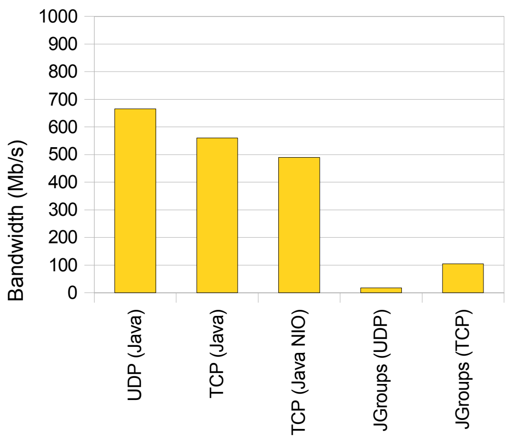


Figure 8: Bandwidth usage with competing garbage collector workload.

4.4 Scenario 3: Competing Garbage Collector Workload

As described in Section 3, this scenario adds competing background garbage collector workload, thus being applicable only to Java protocols. As shown in Fig. 8 the degradation of all protocols except the UDP configuration of JGroups is similar to that with the CPU workload. Recall that both workloads were tuned to consume approximately the same amount of CPU, although performing different tasks.

Fig. 9 shows the same results as a fraction of the original maximum achievable with each protocol. This shows that this workload is however highly problematic for the UDP configuration of JGroups, as it is reduced to as little as 6% of its initial capacity. This is more than enough to explain our trouble with the ESCADA Replication Server and should be worrying to anyone using group communication. Recall that this happens with a workload that consumes approximately only 384MB out of 2GB RAM and only one of the two CPU cores available, as shown in Fig. 1(a) and Fig. 1(b), which should be the nominal load expected in many servers.

Finally, Fig. 10 when compared to Fig. 7 shows that although the impact on kernel based TCP/IP of the competing garbage collector workload seems similar to the CPU workload in terms of average bandwidth, it introduces much more variability which may cause additional trouble for timing sensitive applications.

Lessons Learned: These results show that implementing fine grained retransmission protocols in Java and deploying them as a library in large applications leads to disastrous results in terms of throughput stability.

5 Discussion

The experimental evaluation of communication protocols has been addressed by multiple previous projects focusing both the performance and dependability, including a spectrum of real and simulated stressful environments.

Some work focused on performance evaluation [2], measuring latency and throughput of protocols in different situations, targeting the overhead of architectural decisions and of the Java platform. In this paper we show that such results may be somewhat

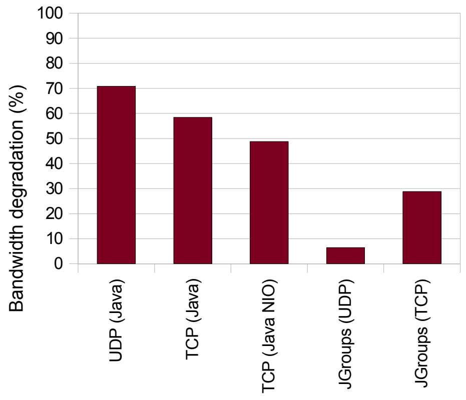


Figure 9: Bandwidth degradation with competing garbage collector workload.

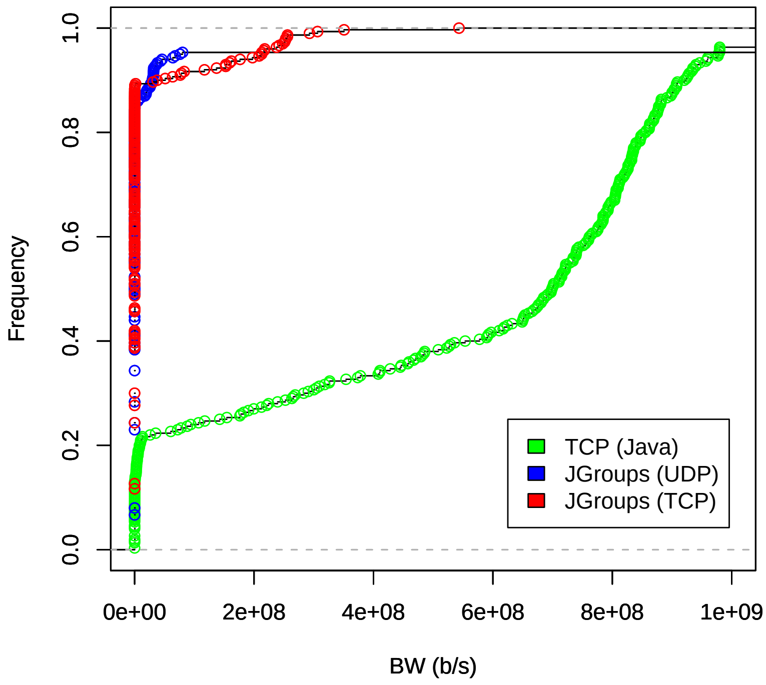


Figure 10: Distribution of bandwidth in time periods with garbage collector workload.

optimistic, since Java-based protocols provided as a library are more vulnerable to competing workloads and thus synthetic benchmarks will not show their actual limits in a real-world environment.

Work focused on JGroups [1] has shown that the performance the TCP-based configuration is superior to the UDP-based configuration. This is attributed to poor performance of the network switch with multicast. In this paper we take this further and show that this is the case even in a point-to-point connection when no multicast is required. Namely, we show that this might be explained also by CPU used by the protocol itself and also by competing workloads on garbage collector and CPU.

An alternative approach has focused on the effect of a slow receiver in the throughput of multicast protocols [4], showing that performance degradation of the group as a whole is unavoidable, even when using state of the art protocol mechanisms. This works was motivated by having observed the degradation in a real setting [13]. This result is highly relevant together with our contribution, showing that our results, measured with just two elements, will have a serious impact in larger groups.

Finally, previous work has targeting group communication protocols with a variety of fault injection techniques, such as memory leaks at the client application level, process hangs, abrupt crashes, and packet loss at the network level [12]. Interestingly, it concludes also that a library-based approach is more susceptible to perturbation. However, we strengthen the result showing that perturbation occurs in normal operational conditions without bugs (e.g. memory leaks) and even in very small groups.

6 Conclusions and Future Work

In this paper we set out to explain the poor performance of group communication protocols observed when there is a competing workload in machines participating in the group. Based on the hypotheses that this effect might be caused by garbage collection and scheduling latency, the first challenge overcome was to reproduce the right workload without having to setup large complex servers.

The benchmark results that were then achieved, comparing multiple protocols with varying competing background workloads lead us to conclude that a protocol with (i) a library-based design, (ii) implemented in Java, and (iii) using an user-level window mechanism on top of UDP, result in a fragile combination that cannot sustain stable high throughput in the presence of a moderate competing background load. Namely, we show a configuration in which throughput is reduced to 6% of that achievable when the system is idle. As a secondary conclusion, we have shown that there is a large performance and resource usage gap between group communication and TCP sockets, that exists even when doing a similar point-to-point communication task.

These conclusions pave the ground for future work in several directions. First, it is interesting to reproduce the results with a wider variety of experimental settings. Namely, using different operating systems and Java virtual machines. For instance, the novel Garbage-First Java garbage collector [7] might have an impact in the results. Second, it is interesting to determine exactly how competing workloads impact the throughput of protocols. Finally, since the gap between application-level protocols and bare TCP is so large, there is definitely room for improvement in the design and implementation of group communication protocols that should be explored.

References

[1] T. Abdellatif, E. Cecchet, and R. Lachaize. Evaluation of a group communication middleware for clustered J2EE application servers. In *Proceedings of the 2004 International Symposium on Distributed Objects and Applications (DOA)*, pages 1571–1589, 2004.

[2] R. Baldoni, S. Cimmino, C. Marchetti, and A. Termini. Performance analysis of java group toolkits: A case study. In *International Workshop on Scientific Engineering for Distributed Java Applications*, pages 49–60, London, UK, 2003. Springer-Verlag.

[3] Bela Ban. Design and implementation of a reliable group communication toolkit for java. Technical report, Dept. of Computer Science, Cornell University, 1998.

[4] K. Birman. A review of experiences with reliable multicast. *Software Practice and Experience*, 29(9), July 1999.

[5] N. Carvalho, J. Cachopo, L. Rodrigues, and Rito Silva; A. Versioned transactional shared memory for the fenixedu web application. In *Proceedings of the Second Workshop on Dependable Distributed Data Management (in conjunction with Eurosys 2008)*, Glasgow, Scotland, March 2008. ACM.

[6] V. Cerf, Y. Dalal, and C. Sunshine. Specification of Internet Transmission Control Program. RFC 675, December 1974.

[7] D. Detlefs, C. Flood, S. Heller, and T. Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, New York, NY, USA, 2004. ACM.

[8] A. Correia Jr., J. Pereira, and R. Oliveira. AKARA: A flexible clustering protocol for demanding transactional workloads. In *Proceedings of the International Conference on Distributed Objects, Middleware and Appocations (DOA)*, 2008.

[9] Lime Wire LLC. LimeWire. http://wiki.limewire.org/, 2009.

[10] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 707–710, Apr 2001.

[11] L. Ong and J. Yoakum. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May 2002.

[12] S. Pertet, R. Ghandi, and P. Narasimhan. Group communication: Helping or obscuring failure diagnosis? Technical report, Carnegie Mellong University, 2006.

[13] R. Piantoni and C. Stancescu. Implementing the Swiss Exchange Trading System. In *IEEE International Symposium on Fault-Tolerant Computing*, June 1997.

[14] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[15] Lee Salzman. ENet. http://enet.bespin.org/, 2009.

[16] Spread Concepts LLC. The Spread Toolkit. http://www.spread.org/, 2009.

[17] Transaction Processing Performance Council. TPC-C. In *TPC*, 2009.

[18] Dizan Vasquez. JeNet. https://jenet.dev.java.net/, 2009.

[19] Dag Wieërs. Dstat. http://dag.wieers.com/home-made/dstat/, 2009.