Java for High Performance and Distributed Computing

Mark A. Baker, Aamir Shafi, and Matthew Grove
University of Reading, UK
{mark.baker@computer.org}

Abstract

The Java language first came to public attention in 1995; very soon after it was being speculated that Java may be a good language for parallel and distributed computing. Its core features, including being objected oriented and platform independence, as well as having built-in network support and threads, has encouraged this view. Today, Java is being used in almost every type of computer-based system, ranging from sensor networks to high performance computing platforms, and from enterprise applications through to complex research-based simulations. In this paper we first explore the pros and cons of Java for High Performance and Distributed Computing. Then we outline some software that is actively being used to support high-performance and distributed applications.

1. Introduction

Java [1] is a modern, object-oriented language based on open, public standards. Objects allow a degree of modularity, which makes them easier to understand and maintain, also the paradigm makes it possible to distribute code with a consistent, public API, while keeping the implementation details private. The core Java API is extensive and includes standard packages for threads, sockets, Internet access, security, graphics, sound, and other useful functions. This means, for example, that Java programs which use these standard packages, can execute unchanged on heterogeneous platforms.

1.1 The Pros and Cons of Java

The Advantages of Java

One of the reasons that Java has been taken up so rapidly is it overall simplicity. No programming language is particularly simple, but Java is considered a simple and easy to use object-oriented language when compared to other popular languages, such as C++ or C. Partially modelled on C++, Java has replaced the complexity of multiple inheritance with a structure called an interface, and also has eliminated the use of pointers, which removes the possibilities of a multitude of errors. In Java, memory management is automatic, and many errors, such as buffer overflows, and stray pointers are impossible. Another reason why Java is considered simpler than C++ is because Java uses automatic memory allocation and garbage collection, whereas C++ requires the programmer to allocate memory and reclaim memory.

Java is considered more reliable, as it has integrated exception handling, which deals with error conditions systematically and forces the programmer to take the necessary action to handle errors. Exception handlers can be written to catch a specific exception such as number format exception, or an entire group of exceptions by using a generic exception handler. Any exception not specifically handled within a Java program are caught by the Java run time environment itself. C has essentially no runtime error checking and memory allocation/retrieval is manual

Java has a mature security model, which has been extensively tested by the community at large. At its core, the Java language itself is type-safe and provides automatic garbage collection, enhancing the robustness of application code. A secure class loading and verification mechanism ensures that only legitimate Java code is executed. Today, a large set of application programming interfaces (APIs), tools, and implementations of commonly used security algorithms, mechanisms, and protocols. This provides the developer a comprehensive security framework for writing applications, and also provides the user or administrator a set of tools to securely manage applications.

Java compilers, interpreters, and runtime systems have come a long way too. Today, the execution of well-written Java code can now be on a par with well written C or C++ code. Most Java code is executed by a JVM (Java Virtual Machine), which can be an interpreter, a JIT (Just-In-Time) compiler, or an adaptive optimising system such as HotSpot. Java applications can execute with little or no change on multiple hardware platforms where a compliant JVM exists. This is a compelling argument for using Java, as it obviates the heterogeneous nature of distributed systems and promotes the ideal of “write once, run anywhere”.

Java has built in support for threading. Threads were designed into the language from the start, they are simple to use, and increasingly needed with the rapid take-up of multi-core processes. With threading comes the need to provide concurrency control, which should prevent race conditions, interference and deadlock. Java has comprehensive support for general-purpose concurrent programming, such as task scheduling, concurrent collections, atomic variables, synchronizers, locks, and nanosecond-granularity timing.

When Java applications create objects, the JVM allocates memory space for their storage - when the object is no longer needed the memory space can be reclaimed for later use. Garbage collection in Java operates incrementally on separate generations of objects rather than on all objects every time. The latest version of Java adds the ability to customise the way object memory is recovered, and this helping dispel the idea that interpreted languages are slow.

Java was designed to be “Internet” aware, and to support network programming with built-in support for sockets, IP addresses, URLs and HTTP. Java native includes support for more interesting protocols including Remote Method Invocation (RMI), and those found in CORBA and Jini.

Java has built-in support for comment-based documentation. The source code file can also contain its own documentation, which is stripped out and reformatted into HTML via a separate program `javadoc`. This way API documents can be created, and this is a boon for documentation maintenance and use.

The performance of Java-based applications depends on a number of factors, including coding efficiency, version of the JVM, underlying Operating System, memory available. Java is now nearly equal to (or faster than) C++ on low-level and numeric benchmarks. This is not a shock really as Java is a compiled language, via the JIT compiler.

There are a huge number of Java development tools, for example the Eclipse platform [2], as well as a large number of open source and free software that has been made available by the community of programmers. If nothing else, this software can be a starting place to develop new ideas and more sophisticated applications.

Finally programmer productivity is believed to be at least two times greater with Java. A lot can be done in a short amount of time with Java because it has such an extensive library of functions already built into the language, integrated development environments, and a wide selection of supporting tools.

The Disadvantages of Java

With regards to memory management, there are no destructors in Java. There is no "scope" of a variable per se, to indicate when the object’s lifetime is ended – the lifetime of an object is determined instead by the garbage collector, The `finalize()` method is called by the garbage collector and is supposed to be responsible only for releasing "resources”, such as open files, sockets, ports, URLs. All objects in C++ will be (or rather, should be) destroyed, but not all objects in Java are garbage collected. The Java garbage collector can be changed, but no explicit control over object collection.

Although arrays in Java look similar, they have a very different structure and behaviour in Java than they do in C/C++. There is a read-only length member (size of array) and run-time checking throws an exception if you go out of bounds. In Java a two-dimensional array is an array of one-dimensional arrays. Although may expect that elements of rows are stored contiguously, one cannot depend upon the rows themselves being stored contiguously. In fact, there is no way to check whether rows have been stored contiguously after they have been allocated. The possible non-contiguity of rows implies that the effectiveness of block-oriented algorithms may be dependent on the particular implementation of the JVM as well as the current state of the memory manager.

There is a floating-point issue because it is required that Java programs produce bitwise identical floating-point results in every JVM. This ideal inhibits efficient floating-point processing on some platforms. For example, it eliminates the efficient use of floating-point hardware on processors that utilise extended precision in registers.

Java has a problem with accessing resources outside the JVM, such as directly accessing hardware. Java solves this with native methods (JNI) that allows calls to functions written in another language (currently only C and C++ are supported). Thus, you can always solve a platform-specific problem (in a relatively non-portable fashion, but then that code is isolated). However, this approach does not comply with the "write once run anywhere" philosophy of Java and breaks the programming model because there is no way to ensure code type safety. Also, there are performance overhead in JNI, especially for large messages, due to copying of the data from the JVM's heap onto the system buffer. JNI also may lead to memory leaks because in C the programmer is responsible for allocating and freeing the memory. Finally, accessing languages that are not C/C++ requires a C/C++ wrapper to interact with other languages such as Fortran or Delphi.

In the next sections we describe two Java-based system that support High Performance and Distributed applications.

2. MPJ Express: A High Performance Java Messaging System

Since its introduction in the early 1990s, the Message Passing Interface (MPI) [3] has now become the *de facto* standard for writing HPC applications on clusters and MPP systems. The MPI community has adopted relatively conventional languages like C and Fortran, which is largely a matter of economics as creating entirely new development environments that match the standards programmers expect today is expensive, and contemporary parallel architectures predominately use off-the-shelf micro-processors that can best be exploited by off-the-shelf compilers.

There have been a number of Java-based messaging systems developed over the last decade, for example [4][5][6][7][8]. These systems have often used different messaging mechanisms, ranging from Java Sockets and Remote Method Invocation (RMI), through to interacting with the underlying MPI system using Java Native Interface (JNI), or using proprietary messaging APIs. Experience gained with earlier Java messaging systems suggests that there is no "one size fits all" approach. The reason is that applications implemented on top of these systems can have different requirements. For some, the main concern could be portability, while for others high-bandwidth and low-latency communication. For some applications the main concern could be portability, while for others high-bandwidth and low-latency.

To address the contradictory issues of portability and high-performance; a Java messaging system needs to provide various communication devices. For example, a Java NIO [9] based device, would satisfy the portability requirements by following a pure Java approach. A specialised device could provide high-performance by interacting with Myrinet [10] or QSNet [11] high-speed interconnects.

With the emergence of commodity SMP systems, and more recently multi-core processors, further requirements for messaging software have emerged. New messaging system needs to support inter-node and intra-node messaging to take advantage of this type of architecture. Currently, the most popular way of efficiently programming SMPs and multi-core processors is using thread-based programming. One of the stronger features of the Java programming language is the built-in support for multi-threading that can be exploited on multi-core processors for thread-level parallelism. To support such thread-level parallelism without any constraints, it is important to have a thread-safe Java messaging software.

2.1 MPJ Express

We have developed MPJ Express [12], a thread-safe Java messaging system, which conforms to MPI-like API based on MPJ [13]. An important contribution of MPJ Express is that it can handle nested parallelism on multi-core processors and SMP systems by using message passing and thread-level parallelism. MPJ Express addresses the contradictory issues of high-performance and portability by providing communication devices using Java NIO (pure Java) and Myrinet. It is possible for end users to switch communication protocols at runtime.

MPJ Express has a layered design that allows incremental development, and provides the capability to update and swap layers in or out as needed. This helps mitigate the contradictory requirements of end users because they can choose to use proprietary network devices or choose the pure Java ones that use sockets. Figure 1 shows the MPJ Express layered design, including the MPJ API, high-level, base-level, *mpjdev*, and *xdev* layers.

MPJ Express currently provides two communication devices that are used to implement the basic point-to-point messaging. These devices include a Java NIO based device called `niodev` and Myrinet eXpress (MX) based device called `mxdev`. The higher levels of the MPJ Express software, like the point-to-point and collective communication layers, rely on these devices for their functionality.

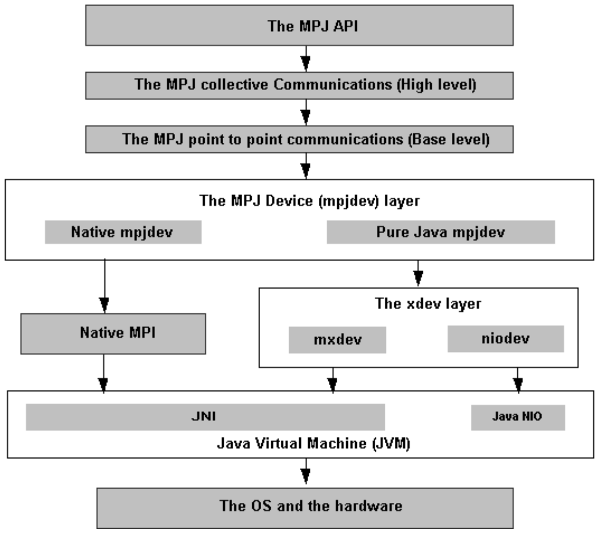
The MPJ API
The MPJ collective Communications (High level)
The MPJ point to point communications (Base level)
The MPJ Device (mpjdev) layer
Native mpjdev
Pure Java mpjdev
Native MPI
The xdev layer
mxdev
niodev
JNI
Java NIO
Java Virtual Machine (JVM)
The OS and the hardware

Figure 1: The Layered Architecture of MPJ Express

The implementation of the `xdev` device layer that provides communication using the Java NIO package is called `niodev`. The standard `java.io` package does not support non-blocking I/O. This means that a new thread has to be started to provide MPI-like non-blocking communication. The Java NIO package solves this problem by providing non-blocking communication. Whenever there is something to read from a particular socket channel, the NIO selector generates a matching read event, which can read the message. This concept is similar to `select()` in C, which helps scalable and efficient I/O. In `niodev`, every process connects to every other process through two NIO channels; we use a blocking channel for writing messages and non-blocking channel for reading messages. There is a separate lock (per destination) associated with each write channel, which means every thread that tries to write a message first acquires the associated lock. No lock is required for reading messages because only one thread receives messages. To implement various send modes - defined by the MPI standard document - `niodev` implements two communication protocols: eager send and rendezvous.

Our Myrinet device called `mxdev` uses JNI to interact with the MX library. It does not implement any communication protocols because the MX library has internally implemented these protocols. Because our buffering API (`mpjbuf`) [13] can use direct byte buffers, we have been able to avoid one of the main overheads of using JNI - copying data between the JVM and the OS. The NIO package makes it is possible to avoid data copying overhead of JNI by using direct byte buffers. In `mxdev` we retrieve the address of the direct byte buffer in the native C code by using the `GetDirectBufferAddress()` method. This method returns the starting address of the memory region referenced by the direct `ByteBuffer`.

A challenging aspect of implementing Java messaging is providing an efficient intermediate buffering layer. The low-level communication devices and higher levels of the messaging software use this buffering layer to write and read messages. The heterogeneity of these low-level communication devices poses additional design challenges. For proprietary networks like Myrinet and QSNet, NIO provides a viable option because it is now possible to get memory addresses of direct ByteBuffers, which can be used to register memory regions for DMA transfers. Using direct buffers may eliminate the overhead incurred by additional copying when using JNI with JVMs that do not support pinning.

We have designed an extensible buffering layer called `mpjbuf` for MPJ Express. This buffering layer allows various implementations based on different storage mediums like direct or indirect ByteBuffers, byte arrays, or memory allocated in the native C code. The higher levels of MPJ Express use the buffering layer through an interface. This implies that functionality is not tightly coupled to the storage medium. The motivation behind developing different implementations of buffers is to achieve optimal performance for lower level communication devices. The buffering layer developed provides variants of write and read methods. It also supports gather and scatter functionality that provides the basis of support for MPI-like derived datatypes. Implementing these features in a Java messaging system is fairly unique because derived datatypes were introduced in the MPI standard for languages like C and Fortran. The derived datatypes can be used for efficient communication of non-contiguous sections of user data. Also, using derived datatypes helps avoid the overheads of Java object serialization and de-serialization.

An important component of a messaging system is the mechanism used for bootstrapping processes across various heterogeneous host nodes. The MPJ Express distribution provides scripts for Windows and Linux that can be used to start the daemon services on compute-nodes. It also allows applications to be executed using class files in an open directory structure, or bundled as a JAR file.

We have compared the performance of MPJ Express, with other messaging systems – elsewhere [15]. In summary the performance evaluation of MPJ Express, against other messaging systems shows that it can achieve good performance but also revealed a performance overhead, which can be classified as an API design issue. With the emergence of Java NIO, the `mpiJava` or MPJ API should be extended to support communicating data directly to and from ByteBuffers or higher abstraction buffers like `mpjbuf.Buffers`.

The emergence of SMP and multi-core processors clusters has raise the need for new programming models, which should provide a portable and efficient solution to nested parallelism without introducing any constraints. We use the term-nested parallelism to signify using multi-threading and messaging to program SMP and multi-core systems. As mentioned before, a main design goal of MPJ Express has been to develop thread-safe communication, which means that the computation and communication within a process can be parallelised on a per-thread basis. Such fine-grained parallelism based on threads may for example, be achieved by using an implementation of OpenMP [16], or by using Java threads on their own. The approach we have taken is to use both OpenMP and Java threads to parallelise the computation and communication. This approach allows application users to freely use hybrid code based on OpenMP and MPJ Express. Application developers are also free to use Java's built-in multi-threading, which may be preferable for programmers who are more familiar with Java threads than the OpenMP standard.

The approach we took was to write a hybrid application using MPJ Express and JOMP [17]. Later this file containing hybrid code with extension `.jomp` was translated to `.java` extension and compiled to produce class files. To execute JOMP-based applications with our runtime, we passed a command line switch `jomp.threads` and specified JOMP's JAR file to `mpjrun`. We did not encounter any fundamental problems in using JOMP threads within a MPJ Express process. In addition, we managed to use JOMP with the latest JDK version. However, we made some important optimisations to the JOMP software. One of these included replacing the busy-wait style implementation of the barrier method, which resulted in high CPU use on a node, with a new barrier implementation. The new implementation provides an alternative to the default four-way tournament barrier originally implemented for JOMP. The JOMP library starts a team of threads at the start of the execution. The master thread is responsible for executing serial parts of the code. During this time the worker threads are doing a barrier waiting for the master thread to reach a parallel region and call the barrier.

2.2 Gadget-2

To help establish the practicality of real scientific computing using message passing Java we ported Gadget-2 [18] to Java using MPJ Express from scratch. The Java version was developed as an experiment to help us understand where Java stands in comparison to C. In addition, the Java version is a test case to gauge the performance of MPJ Express in a real-world application. We also tested our ideas of introducing nested parallelism and measured the performance gains.

Gadget-2 is a free production code for cosmological N-body and hydrodynamic simulations. The code is written in the C and parallelised using MPI. It simulates the evolution of very large, cosmological-scale systems under the influence of gravitational and hydrodynamic forces. The universe is modelled by a sufficiently large number of test particles, which may represent ordinary matter or dark matter. The main simulation loop increments time steps and drifts particles to the next time step. This involves calculating gravitational forces for each particle in the simulation and updating their accelerations. We are particularly interested in the parallelisation strategy, which is based on an irregular and dynamically adjusted domain decomposition, with copious communication between processors.

The original C version of Gadget-2 was manually translated to the Java. We deliberately kept similar data structures in the translated version, so that we could cross reference the original source code for debugging. Currently there are some functional limitations compared with the C version. For example, the Java version only provides the option of using BH Oct tree for calculating gravitational forces. For communication, of course we use MPJ Express. Gadget-2 extensively uses the point-to-point and collective MPI methods.

The biggest simulation that we carried out with the Java version contained 56 million particles on 16 nodes - each MPJ Express process contained roughly 3.5 million particles. We carried out the tests on a larger cluster called NW-GRID located at the Daresbury Laboratory, UK. The system consists of 96 nodes divided into three racks, each containing 32 nodes with 2 dual core 2.4 GHz AMD Opteron 64-bit processors. Each node has Gigabit and is running SuSE GNU/Linux with kernel 2.6.11.4-21.11-smp. The C compiler used in GCC 3.3.5 with support for 64-bit processor and POSIX threads enabled. JDK 1.5 update 7 was used to compile and run the Java code. The JDK used is specialised for AMD Opteron 64-bit processors and the virtual machine used was Java HotSpot 64-Bit Server VM. The evaluation presented is based on first hundred time steps of the simulation, which is 10% of the total simulation.

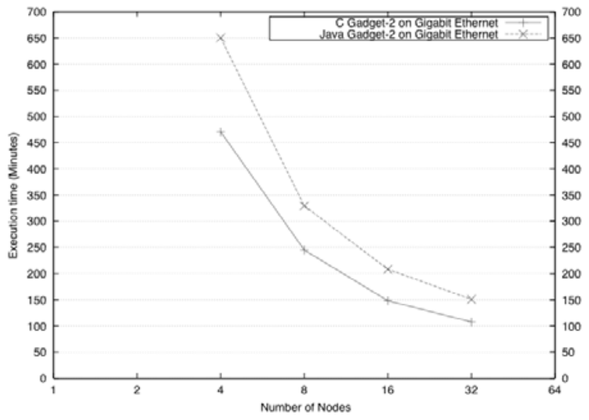

Figure 2: A Comparison of C and Java versions of Gadget-2

Figure 2 shows the total execution time of the C Gadget-2 and Java Gadget-2 code. It can be seen that the Java version achieves comparable performance, with 30% or less performance overhead relative to the C version. A performance comparison of the Java and JOMP Gadget-2 version is shown in Figure 3. As each node contains 2 dual core Opteron processors we ran 4 JOMP threads in each MPJ Express process to exploit each core or CPU efficiently in a node. The two versions ran custom initial conditions with 2 million particles with the Periodic Boundary Conditions (PBCs) were turned on to increase the tree walk time. We expect to see large performance gains in this case because we have extensively used thread-parallelism for tree walks. The Java version with four JOMP threads clearly shows the advantages of our approach of using thread parallelism. Using JOMP threads has increased the performance of the simulation significantly. The overall execution time has been reduced by a factor of 2 to 3, depending on total number of processors used. Due to limitations of time we were unable to benchmark this simulation with versions of the code, which use multiple MPI processes on a node. We aim to do this in future.

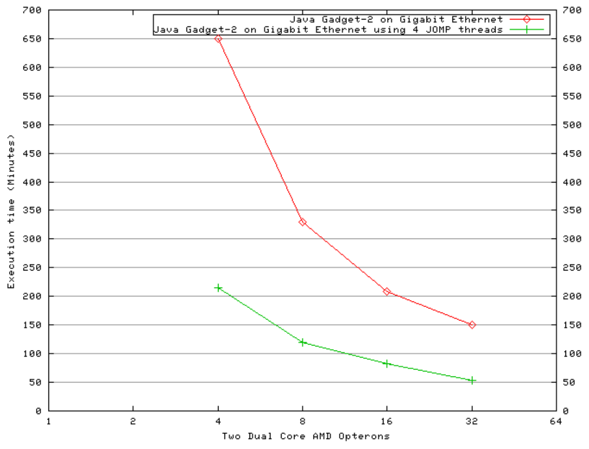

Figure 3: The Execution Time for Single and Multi-threaded Java Gadget-2 for the Cluster Formation Simulation on NW-GRID Cluster.

2.3 MPJ Express Summary

We have designed and implemented a new Java messaging software called MPJ Express. This messaging system coupled with Java or JOMP threads can help efficiently program parallel applications on next-generation systems. A unique feature of MPJ Express is that it provides thread-safe communication devices that allow multiple threads in an application to communicate safely. MPJ Express is currently available for download as free and open-source software. James Gosling, one of the founders of Java, called MPJ Express one of his favourite MPI-like library in his Blog “MPI Meets Multicore” [19].

To help establish the practicality of scientific computing using a Java message passing system we ported Gadget-2 to Java using MPJ Express. Gadget-2 is a massively parallel structure formation code. Versions of the original C code have been used in the so-called “Millennium Simulation” that evolves ten billion dark matter particles from the origin of the universe to the current day. The performance evaluation of the Java version revealed that it could achieve comparable performance to the original C code. The performance of Java Gadget-2 reinforces our belief that Java is a viable option for HPC. With careful programming, it is possible to achieve performance in the same general ballpark as the C code.

3. Tycho: A Wide-area Messaging Framework with an Integrated Virtual Registry

In any distributed environment the various remote entities need a means to publish their existence so that clients, needing their services, can search and find the appropriate ones. The publication of information is via a registry service, and the interaction is via a high-level messaging service. Typically, separate libraries provide these two services. Tycho [20] is an implementation of a wide-area asynchronous messaging framework with an integrated distributed registry. This integrated software frees developers from the need to assemble their applications from a range of potentially diverse middleware offerings, which simplifies and speeds application development and more importantly allow developers to concentrate on their own domain of expertise.

3.1 Tycho's Architecture

Tycho is a Java-based framework based on a Service Oriented Architecture (SOA) that uses a publish, subscribe and bind paradigm. We have used an architecture similar to the Internet, where every node provides reliable core services, and the complexity is kept to the edges. This implies that the core services can be kept to the minimum needed, and endpoints can provide higher-level and more sophisticated services, that may fail, but will not cause the overall system to crash. The design philosophy for Tycho has been to keep its core relatively small, simple and efficient, so that it has a minimal memory foot-print, is easy to install, and is capable of providing robust and reliable services. More sophisticated services can then be built on this core and are provided via libraries and tools to applications. This will ensure Tycho is flexible and extensible so that it will be possible to incorporate additional features and functionality later. Tycho's functionality has all been incorporated within a single Java JAR with the only requirement being a Java 1.5 JDK for building and running Tycho-based applications.

Tycho consists of the following components:

- Mediators that allow producers and consumers to discover each other and establish remote communications,
- Consumers that typically subscribe to receive information or events from producers,
- Producers that gather and publish information for consumers.

In Tycho, producers and/or consumers (clients) can publish their existence in a directory service known as the Virtual Registry (VR). A client uses the VR to locate other clients, which act as a source or sink for the data they are interested in. The VR is a distributed service provided by a network of mediators. Where possible, clients communicate directly, however, for clients that do not have direct access to the Internet, the mediator provides wide-area connectivity by acting as a gateway or proxy into a localised Tycho installation.

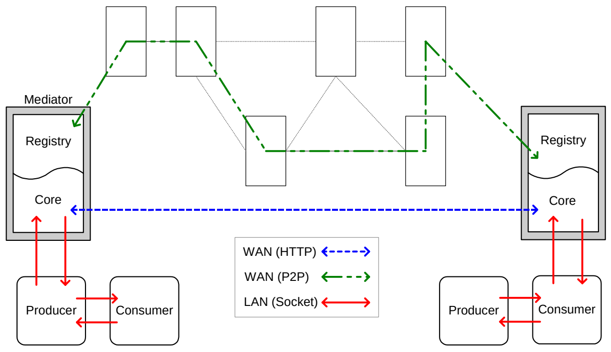

Figure 4: A View of Tycho's Architecture

Figure 4 shows Tycho clients communicating between two remote sites connected via the Internet. The Tycho VR is made up of a collection of services that provides the management of client information and facilitates locating and querying remote Tycho installations. A client registers with a local mediator, part of the VR, when it starts-up. The VR provides a locally unique name for each client and periodically checks registered entities to ensure their liveliness, removing stale entries if necessary. The VR consists of the following components:

- The transport handler allows different protocols to be used between Tycho components. Currently, TCP sockets, SSL, HTTP(S), and Internet Relay Chat are supported.
- The local store provides an abstract interface to a mediator's information store. The store is implemented using a variety of data storage technologies, including a JDBC-based storage medium and an in-memory data structure (simple store). JDBC permits the use of a range of SQL storage technologies ranging from Oracle to MySQL.
- The query parser and result annotator components translate queries and responses into an intermediate internal format in order to allow Tycho to support different query languages and

permit interoperability with other systems in the future. Tycho currently supports a subset of the ANSI-SQL query language and LDIF [21] as a response mark up.

Tycho's VR provides information for uniquely identifying a client, URLs that are used by the transport handlers to locate and communicate with a client and a schema field, which can be used to store information about the capabilities of a producer or consumer.

Security is an essential requirement for any distributed system. Tycho's architecture is designed to support both encryption and access control to provide a secure environment. Encryption is provided at the transport handler level using SSL to encrypt messages sent via the HTTP, Socket and IRC handlers. Access control is provided using a layered approach. In keeping with the design philosophy of Tycho, we re-use existing infrastructure. Access control is can be via the use of a proxy server, or the security features of an IRC daemon.

3.2 Performance Tests

A performance study of Tycho against similar systems has been made. For the purposes of evaluating Tycho's messaging performance, comparative tests were made with the NaradaBrokering [22] system and the performance of Tycho's virtual registry was compared to Globus MDS4 [23] and to R-GMA [24], further details can be found elsewhere [25], a summary is provided next.

When looking at point-to-point performance, on a LAN for messages less than 2 Kbytes, Tycho and NaradaBrokering have comparable performance. Tycho achieves 95% of the maximum bandwidth, whereas NaradaBrokering uses 65.3%. Overall, the performance of the two systems is similar. Tyco's current performance is inhibited by the fact that it creates a new socket for each message send, whereas NaradaBrokering reuses sockets instances once they have been created. Incorporating such as scheme in Tycho will further reduce its latency.

The scalability tests show that Tycho and NaradaBrokering producers and consumers are stable under heavy load, although performance is weaker when there is a large ratio of consumers to producers. The heap size for NaradaBrokering becomes a limiting factor in circumstances where a broker is receiving messages faster than it can send them, as the internal message buffer fills until the heap is consumed. The Tycho tests were performed without modifying the heap size, as the use of new sockets per message automatically throttles performance and prevents messages from being received faster than they can be sent.

We tested Tycho against R-GMA and MDS4 in order to show that our philosophy of keeping the core functionality as simple as possible yields performance gains over these systems while still supporting the registry functionality required. In Tycho, more complex functionality is added to the edge of the implementation rather than by increasing the complexity of the core, thus is it is essential that the core perform well. Tycho,

When testing the affect of the number of records on response time, we see that when selecting a single record from 100,000, Tycho responds 32 seconds faster than R-GMA. MDS4 runs out of heap space for larger records sizes, which suggests that they should look at either storing the data more efficiently or moving to a file-backed store that is not limited by heap size. The performance tests show Tycho's VR had a lower response latency than R-GMA and MDS4. With 100 clients Tycho was 94 seconds faster than R-GMA and 65 seconds faster than MDS4. The results highlight that one of the strengths of Tycho is its performance under load. Tycho's performance is linear with regard to both increasing numbers of clients and response sizes.

3.3 Example Applications

Tycho is being used in several projects including GridRM [26], Slogger [26], and the VOTechBroker [27], which is part of the European Virtual Observatory project [28].

The Tycho swarm utility provides generic content distribution. The main bottleneck in traditional client-server content distribution using a system, such as a Web server and browser, is the bandwidth available to the server. In order to reduce the impact of this bottleneck, swarm downloading makes use of the upload bandwidth of the clients downloading content, as well as the bandwidth of the servers. A popular implementation of this peer-to-peer content distribution system is BitTorrent [29], which is a file sharing protocol. In a swarm content distribution system each participant is called a peer. A peer with a complete copy of the content is called a seed. Together all participating peers and seeds are called a swarm.

Content is broken into multiple pieces of an arbitrary size, commonly called chunks. Peers request chunks from all other peers participating in the swarm. At the start of the publishing, only the seeds will have chunks available, but when a peer downloads a chunk it makes it available to the rest of the swarm. In this way the bandwidth resources of the entire swarm are utilised to distribute the content rather than just the bandwidth from the seeds. This can lead to considerable speed-ups when there are a large number of peers, or seed bandwidth is limited. In this type of system there needs to be a mechanism for peers to locate each other; in the current version of BitTorrent a central tracker is used.

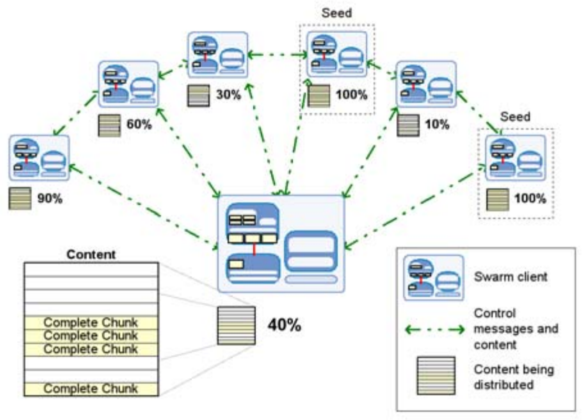

Figure 5: A high-level of Tycho's swarm utility. Content is being distributed among a swarm of seven peers containing two seeds

Figure 5 shows a Tycho swarm participating in content distribution. The utility implements peers and seeds as a single Tycho client, which acts as a producer and consumer simultaneously. When content is published using a Tycho swarm, the seed client registers all of the chunk information into the Tycho VR. Clients then query the VR to request a list of the locations of chunks that are available. Clients can then request and receive chunks using Tycho messages. When a client successfully downloads a chunk it updates its entry in the VR to say it now has the chunk available. When a client runs out of available chunks and has not completed the content download, it queries the VR requesting additional locations for any chunks that it does not have. This is possible because of the fine-grained queries supported by Tycho, which means that the addresses of all peers, which have one or more chunks, can be located using a single SQL query. If a client exhausts all of the sources for chunks when it sends a chunk to another client, it appends a request to be notified when the client has new chunks available. When a client receives this request it re-queries the VR. An MD5 hash uniquely identifies each file, which is available for download. The records in the VR describing each available file contain the MD5 hash, the original file name and the total size of the content.

A number of performance tests showed that the implementation provides higher performance than client-server content distribution. The tests published files at least up to 100 Gbytes in size and that performance scales with number of peers.

3.4 Tycho Summary

Tycho's has a relatively small, simple and efficient core, so that it has a minimal memory footprint, is easy to install, and is capable of providing robust and reliable services. More sophisticated services can then be built on this core and are provided via libraries and tools to applications. This provides us with a flexible and extensible framework where it is possible to incorporate additional feature and functionality, which are created as producers or consumers, and do not affect the core. Tycho's functionality has all been incorporated within a single Java JAR and requires only Java 1.5 JDK for building and running applications.

Tycho performance is comparable to that of NaradaBrokering, a more mature system. Whereas, compared to MDS4 and R-GMA, Tycho shows superior performance and scalability to both these systems. In addition, we would argue that both MDS4 and R-GMA have problems with memory utilisation and without significant extra effort limited scalability. Additional APIs and specifications can be easily incorporated into Tycho by simply creating compliant producers and consumers. An important advantage of Tycho's architecture is that addition of further producers/consumers will not affect its core, or existing producers/consumers.

4. Overall Summary and Conclusions

In this paper we first discussed the pros and cons of Java as a language and technology for developing High Performance and Distributed systems and applications. We then moved on to describe a Java messaging system (MPJ Express) that supports parallel applications, and then a system (Tycho) that provides asynchronous messaging and an integrated registry that can be used for a range of distributed applications.

Java clearly encourages better software engineering by promoting object-oriented programming and by providing a system that allows programs to be executed anywhere that a compliant JVM exists. Java has many extra safety features including array bounds checking that could help identify potential bugs in the code. We found, for example, that in the original Gadget-2 code a seventh element of a six-element array was accessed. The Java Gadget-2 helped identify the error by throwing an ArrayOutOfBound exception.

References

[1] Java, http://www.java.com

[2] Eclipse, http://www.eclipse.org/

[3] MPI, http://www-unix.mcs.anl.gov/mpi/

[4] M. Baker, D. Carpenter, G. Fox, S. Ko and X. Li, mpiJava: A Java MPI Interface, First UK Workshop, Java for High Performance Network Computing at EuroPar 1998, September 1998, http://www.cs.cf.ac.uk/hpjworkshop/

[5] Glenn Judd, Mark Clement, and Quinn Snell, DOGMA: Distributed Object Group Management Architecture, ACM 1998 Workshop on Java for High-Performance Network Computing, Concurrency: Practice and Experience, 10(11-13), 1998

[6] K. Dincer, jmpi and a Performance Instrumentation Analysis and Visualization Tool for jmpi, First UK Workshop, Java for High Performance Network Computing at EuroPar 1998, September 1998, http://www.cs.cf.ac.uk/hpjworkshop/

[7] Steven Morin, Israel Koren, and C. Mani Krishna, Jmpi: Implementing the message passing standard in Java, Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS), CS Press, page 191, 2002

[8] William Pugh and Jaime Spacco, MPJava: High-Performance Message Passing in Java Using Java.nio, Proceedings of Workshops on Languages and Compilers for Parallel Computing, pages 323–339, 2003

[9] Java NIO, http://javanio.info/

[10] Myrinet, http://www.myri.com/

[11] QSNet, http://www.quadrics.com/

[12] MPJ Express, http://mpj-express.org

[13] Bryan Carpenter, Vladimir Getov, Glenn Judd, Anthony Skjellum, and Geoffrey Fox, MPI for Java: Position Document and Draft Specification, Technical report, Java Grande Forum, November 1998, http://www.javagrande.org/reports.htm.

[14] M.A. Baker, D.B. Carpenter and Aamir Shafi, A Buffering Layer To Support Derived Types And Proprietary Networks For Java HPC, the international Journal of Scalable Computing - Practice and Experience, 2006, ISSN 1895-1767

[15] M.A. Baker, Bryan Carpenter, and Aamir Shafi, MPJ Express: Towards Thread Safe Java HPC, Procs of the IEEE International Conference on Cluster Computing (Cluster 2006), Barcelona, Spain, September, 2006, ISSN: 1552-5244

[16] OpenMP, http://www.openmp.org/

[17] JOMP, http://www2.epcc.ed.ac.uk/computing/research_activities/jomp

[18] Gadget-2, http://www.mpa-garching.mpg.de/gadget/

[19] James Gosling's Blog, http://blogs.sun.com/jag/entry/mpi_meets_multicore

[20] Tycho, http://www.acet.rdg.ac.uk/projects/tycho

[21] LDIF, http://en.wikipedia.org/wiki/LDIF

[22] NaradaBrokering, http:// www.naradabrokering.org

[23] Globus MDS, http://www.globus.org/toolkit/mds/

[24] R-GMA, http://www.r-gma.org/

[25] M.A. Baker, and Matthew Grove, Tycho: A Wide-area Messaging Framework with an Integrated Virtual Registry, accepted for publication in a Special Issue on Grid Technology with the International Journal of Supercomputing, 2006, ISSN: 0920-8542

[26] GridRM, http://gridrm.org

[27] VOTechBroker, http://dsg.port.ac.uk/projects/votb/

[28] Virtual Observatory, http://www.euro-vo.org/

[29] BitTorrent, http://www.bittorrent.com/