

The most important feature of enterprise software is the quality of its server technology. Performance and scalability are two very important features of enterprise technology, and frequently some of the hardest to quantify and measure. This whitepaper will describe some of the underlying technology that serves as the foundation of the Rhombus IM platform.

Making a server scale to high levels of demand is not something that can be added at a later time - it needs to be designed in from the beginning. Every design choice must be evaluated from a scalability perspective to ensure that nothing compromises the overall architecture.

Analyzing Server Performance

A traditional Internet server needs to scale in several ways - it needs to handle many simultaneous connections, it needs to handle traffic and requests with low latency, and it needs to minimize resource usage while doing all of the above. It is also imperative to be able to scale across machines. Scaling inside one server process is only half of the battle - a scalable infrastructure must be able to fail over from one server to another and share state between machines.

Connection Scaling

There are two traditional ways to handle a new incoming connection to a server. The first way is to setup a unique thread of execution that blocks waiting for information to be sent over that connection, then handles the information, writes any necessary output, and then continues waiting for more input bytes. This has the advantage of implementation simplicity - a thread need not worry about managing its state because the operating system does that on its behalf. However, it has several severe disadvantages.

A thread is a very costly thing to create, especially on Windows and Linux operating systems. The operating system needs to allocate memory to save the thread's internal state. Switching between threads is also costly, as the outgoing thread needs to save its state to memory, and the incoming thread needs to restore its state back into the processor context. This is known as *context-switching overhead*, and the penalties one pays for such switching grow dramatically as the number of threads increases. This is a particular insidious penalty because it is not noticeable at low connection volumes, but becomes prohibitive as the connection load grows.

As an example, consider one thousand concurrent connections, each requiring a single thread of execution to process its I/O (some implementations are even worse, allocating two or more threads per connection). This results in one thousand threads being allocated at the operating system level. On Windows NT based operating systems, each thread requires 1MB for stack space and local heap storage. A quick back of the envelope calculation reveals that such a setup will consume 1GB in memory overhead. If this weren't bad enough, let's assume that each thread is reasonably busy and that every second we have ten thousand context switches (you can examine this number by using `vmstat` or the Windows Performance Monitor).

According to the article "Context switching," by Dr. Edward Bradford of IBM¹, a context switch on Windows XP takes approximately 25-30 microseconds ($1\mu s = 10^{-6} s$) to complete. For our above example, if we solve for the amount of time taken in switching overhead above, we get:

$$t_{switch} = 10^4 \bullet 3 \times 10^{-5} = .3$$

¹ <http://www-106.ibm.com/developerworks/linux/library/l-rt9/?dwzone=linux#h1>

This implies that 30% of the processor time each second is spent simply switching between threads! Your 1GHz processor just got downgraded to 700MHz.

In contrast, a more scalable approach to handling many simultaneous connections is to restrict the number of worker threads to the number of available processors, and to manage input and output tasks without retaining unnecessary state. This is the approach that the Apache server uses - it is one of the most scalable and robust server products available today. It is also the approach used by the Rhombus IM Server. This approach is called the `select()` approach, named after the operating system function that is used to service multiple I/O requests from a single thread.

In this case, ten thousand connections only require one thread of execution, meaning that all of your processor cycles are available to do meaningful work. It also means that the memory overhead is kept to a minimum - only the necessary socket objects are created on a per-connection basis, instead of the full stack required by a thread. You may wonder why any commercial server product would choose the former approach when it is clearly not scalable. Very simply, it is much harder to create a server that can switch between multiple units of work on a single thread than it is to let the operating system handle this switching. Most companies and most servers take the easy way out, but they are paying a huge performance cost that may not be borne out at the low end of scalability. That is, they can "get away with it," at least for a little while.

In Java, the capability to handle I/O in this way has only recently become available with the advent of Java 1.4's NIO primitives. They are very complicated and most development shops steer clear of these primitives. This is why Rhombus IM Server requires Java 1.4.

Handling Abnormal Conditions

Abnormal conditions are another aspect of server performance. Typical platforms optimize for the "common case," and therefore are unable to handle exceptional conditions and restore functionality to conforming clients. An experiment to try is to "pause" an IM client that is connected to a server. You can do this in UNIX by hitting Ctrl+Z, effectively preventing the client from servicing its I/O. In Windows it's a little harder to accomplish this - you basically have to induce the client to hang (it's easier with some than with others).

Each server has its own way of dealing with a situation like this. Basic server implementations will simply block forever until the client becomes responsive again. This type of optimistic flow control is unacceptable in a production server platform. A single client should never be able to hold the server platform hostage.

It takes up-front design choices to build in reasonable behavior in situations like this. The Rhombus platform offers design-time choices to applications when overflow conditions require drastic action. Out of the box, the Rhombus IM server will ignore the catatonic client and continue to process information for other clients. In other words, no one will notice that a client somewhere is having problems - and this is exactly the desired behavior.

It is possible to customize this behavior as well, so that information can be "expired" when it is no longer relevant, or the server can actually take matters into its own hands by forcibly disconnecting clients that are not responsive. However, such decisions must be made carefully because the balance of power can shift too easily to publishers. Sending information is much easier than processing it, and a rogue publisher can trivially send much more information than a

subscriber can reasonably process. Doing this is effectively a "death ray," because the Rhombus server will terminate the client who cannot keep up with message publishing.

Scaling with Hardware

A great software platform can do a lot to make a single server process handle as much work as possible. Still, any given machine has real physical limitations or operating system level limitations (particularly on Windows platforms) that require you to scale your IM system to multiple machines. Having two or more machines is a requirement if you want to provide true fault-tolerance and redundancy for your infrastructure - if you can't be down.

Many servers simply do not have the capacity to operate consistently across more than one machine. Again, like scalability considerations, it needs to be designed in from early in the development process. It needs to be transparent to users so they don't care if they are on one machine or another. They need to be able to talk to anyone, regardless of what server they are on.

Clustering makes it possible to provide a true load-balanced solution for your IM users. Place a cluster of Rhombus IM servers behind a Cisco Local Director or other similar technology and you're done. You instantly have a redundant, fault-tolerant server installation for your users. If one of the machines goes down, the other will take over.

Conclusions

The Rhombus IM Platform has a variety of scalability advantages over its competition. In this crowded space it is hard to understand what differentiates various IM vendors and why a particular server platform doesn't scale and why another does. Whatever your installation size, you want to make sure that your choice for this critical business technology has the capacity to handle your demands for IM, and can handle the demand as your business and your use of IM grows with time.

The Rhombus IM platform has all of the important scalability features that will help you meet this demand in the future. The combination of single-threaded I/O operation, low memory overhead, data overflow handling and clustering features all guarantee that your Rhombus IM Server installation will handle whatever you, your colleagues and your customers need.