

---

White Paper on

# **Lyris ListManager Extreme**

**A Third Generation Email Engine**



# Contents

- Executive Summary Lyris ListManager Extreme White Paper** **1**
  - Lyris ListManager Extreme: The Third Generation ..... 1
  
- The First Generation: One program, one process** **3**
  - Summary ..... 3
  - How the first servers worked ..... 3
  - Comparison Chart ..... 4
  
- The Second Generation: One thread, one process** **5**
  - Summary ..... 5
  - How the second generation works ..... 5
  - Comparison Chart ..... 7
  
- The Third Generation: Many tasks per thread, and notify when done** **8**
  - Summary ..... 8
  - How the third generation works ..... 8
  - Comparison Graph ..... 10
  
- Notes** **12**
  - Apache vs. sendmail on Unix ..... 12
  - Windows NT lack of forking and Apache, sendmail..... 12
  - Other third generation servers: NaviServer and AOLServer..... 12
  - A note on comparable-measurement standards..... 13



# Executive Summary

## Lyris ListManager Extreme

### White Paper

---

## Lyris ListManager Extreme: The Third Generation

This paper lays out how the architecture of email servers has evolved over three generations, and describes the advantages of the new, third generation architecture adopted by Lyris ListManager as compared to the previous two generations.

The spectacular growth of websites on the Internet has spurred advertisers and marketers to find new ways to bring customers back to their website. Permission-based email advertising has had remarkable success in doing so,

especially in comparison to bulk mail and banner ads. However, as permission-based marketing has exploded, so have the pressures on conventional email servers and email server vendors.

As demand for more efficient volume processing has grown, developers have searched for ways to increase speed while keeping memory and processing within acceptable boundaries. These efforts can be grouped into three “generations” of email processing.

First generation email servers such as sendmail and qmail create a process for each individual message sent. This method works fine for sending volumes of 5,000-10,000 unique emails per hour, an amount formerly thought to be high. But this approach is limited: the demand for memory becomes excessive, and the number of programs that can be run becomes a limiting factor.

The second generation attempts to overcome the limitations of the first by using a technique known as multi-threading. This design uses memory more efficiently, and it is also faster, allowing up to 100,000-300,000 or more unique sends per hour. However, the constraints of this design have

#### Lyris ListManager Extreme

- Troublefree list administration
- Handles high volumes
- Fast, reliable
- Advanced third generation architecture

#### Pros:

- High Speed
- Reduced hardware needs

#### Con:

- Requires high bandwidth

resulted in diminishing returns: even when additional computing and memory hardware are lavished upon systems, the processing architecture breaks down.

The first and second generations are also hampered by the limitations on the receiving end. No matter how quickly an email server could send out messages, it is still constrained by the delays other servers create in receiving mail. Until the receiving process is finished, the first and second generation servers can only sit and wait before moving on to another message.

The burgeoning demand to send greater volumes of email quickly and reliably has led Lyris architects to the third generation of server engines: Lyris ListManager Extreme.

The design breakthrough of ListManager Extreme allows one thread to handle many tasks at once. This new technology uses considerably less memory and processing time than previous generations. Instead of diminishing returns, adding hardware resources leads to linear improvement

Because ListManager Extreme has more processing resources, it can create more open connections while efficiently processing simultaneous sends, a dramatic improvement over what has been possible with email servers from earlier generations. Instead of waiting for receiving email servers to respond, ListManager Extreme can open up many more connections, continue processing messages and then return when notified that the connections are complete. Thus, ListManager Extreme doesn't need to wait while receiving servers are processing its messages to create more connections.

With this third generation of Lyris ListManager, we can open and successfully process up to 5,000 connections—and that's with our first version! We anticipate that by the end of the year 2000, ListManager Extreme will be able to send one million messages per hour.

# The First Generation: One program, one process

## Summary

The advantages of ListManager Extreme as a third generation mail server are best appreciated in comparison to the two generation of servers that have come before it. The next section will explain the first approach programmers took to send email and web pages across the Internet, and how software and hardware constraints limit the speeds this approach can achieve.

---

## How the first servers worked

The first web servers take a straightforward approach to answering requests for web pages: as requests come in for a web page, the web server will return the web page and then move on to the next request. Each request, or task, uses just one process; when the task was completed, the process exits. The benefit of this first generation architecture is that it is simple to understand, modify and maintain.

However, only one request can be handled at a time, and requests could be lost if the backlog grew too large. The solution to this problem is to run many copies of the process at once, with each process handling just one task at a time. Then, a routine similar to a traffic cop accepts all the tasks that came in and hands them off to the waiting processes. The Apache web server takes this approach, and as of the date of this paper (May, 2000), it is still the approach Apache uses in its Unix version. Since the Apache web server for Unix runs on more web servers than any other, this one process per task approach predominantly serves the most web sites today.

Sendmail, the dominant email server on the Internet, takes the same approach as Apache. Sendmail is a free email server included with the Unix operating system. When sending a message to a number of people, a master sendmail process (for techies: forks) creates new sendmail processes, and each sendmail process is responsible for sending a single email message. Like the Apache web server, one process is responsible for one task, which in this case is sending an email message.<sup>1</sup>

Memory and CPU time are the two major limitations of this first generation architecture. In order to handle large numbers of simultaneous tasks, a large number of simultaneous processes must be held in memory at once. Because both Apache and sendmail are large programs, they take a large amount of memory per process. Therefore, a typical computer is unable to run more than a few dozen copies of the application before running out of memory. At that point, performance suffers greatly.

The usual way for the first generation architecture to accommodate larger numbers of simultaneous processes is to add huge amounts of memory to a system. This technique works to a point; however, it is expensive. More importantly, even today there is a practical limit to how much memory that can be added to a system without diminishing returns.

Another way to work around this problem is to make the size of the individual processes smaller through better programming. Sendmail is a large program with many capabilities, but it since it doesn't typically use all of those capabilities each time it sends a message, most of the memory used by the program is wasted. Another free Unix mail server, qmail, increases speed by using less memory than sendmail. Because each qmail process takes less memory, more simultaneous copies of qmail can be run and thus mailing speed can be increased.

Although qmail's approach helps to speed email processing, it remains limited. Even if the memory needed by the first generation architecture is lessened, it remains limited by the amount of CPU time required. Individual processes take quite a bit of CPU time from the system, and there is a maximum number of processes which an operating system can effectively handle. This limitation on processes also sets an upper limit on the speed of this first generation architecture.

---

## Comparison Chart

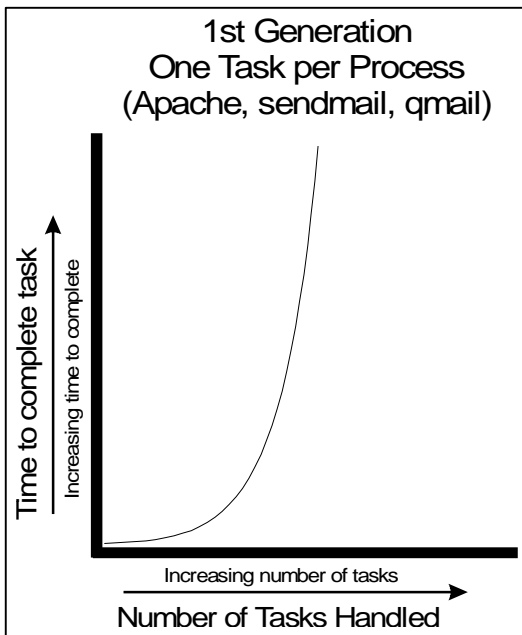


Figure 1

In Figure One, a chart shows the speed and scalability of the first generation web and email servers.

This generation works quite well for low numbers of tasks: all the processes in memory are under the operating system and CPU limits, and thus are handled well.

For a medium number of tasks, the overhead of increasing the number of processes begins to take its toll. As memory becomes scarce, the disk starts to swap, and the operating system becomes increasingly inefficient as the number of processes to juggle grows.

As the number of tasks grows to a large number, this architecture falls apart. The system runs out of memory or creates too many processes, so that the machine is not even capable of responding to the requests in a timely fashion. Web pages will experience timeouts, and email will experience email send timeouts and retries.

Thus, with the first generation architecture there is a fairly low ceiling to the scalability of the application and the current demand for high performance cannot be met.

# The Second Generation: One thread, one process

## Summary

The previous section demonstrated that the first generation of servers are effective at executing a low number of tasks, but are not able to scale to handle a larger number without breaking down completely. The next section details how the second generation of servers attempted to overcome the limitations of the first through the use of multi-threading. Although able to handle more tasks than the first generation, the second generation of servers is also limited by hardware and software constraints.

---

## How the second generation works

In an effort to perform more tasks more quickly than first generation architectures allowed, a number of web and email vendors analyzed the problem with the first generation and came up with a common solution: an advanced operating system method called multi-threading.

Multi-threading is a technique where an individual "thread" can perform a single task, and many threads can run at the same time within a single process. Conceptually, a thread is very similar to a process; each thread completes a task independently, in a separate execution context. In other words, stuff gets done while other stuff gets done.

The fundamental difference between a process and a thread is that a process has an entire copy of the program to itself, and thus consumes quite a bit of memory. A thread is nothing but a separate execution context within a single program: each thread shares all the memory with the other threads, with the exception of just a small amount of per-thread memory so that the operating system can keep track of the execution context of each thread.

What does this difference mean in practice? First, many more threads can run on a given amount of memory than the same number of processes, because each individual thread uses much less memory. Also, operating systems can handle significantly more threads than individual processes

because multi-threading is typically much more efficient. That is, the amount of CPU time per thread is lower than the amount of time per process.

Creating an individual thread is also a much faster and less memory intensive task than creating a new process. Thus, programs which create and destroy threads quickly will be much faster and more efficient than programs that create and destroy processes quickly.

Using individual threads for separate tasks is so much more efficient than using separate processes that Microsoft Windows NT completely lacks the important `fork()` operating system function call, present on all versions of Unix and virtually essential for the first generation web and email servers.<sup>2</sup>

By using this multi-threading approach, Apache on Windows NT is able to handle far more simultaneous requests than Apache on Unix. In our own tests of Apache for both Unix and NT, we have found that servicing ten CGI requests on Unix takes about 150 megabytes of RAM (60mb for the main Apache process, 10mb for individual page Apache processes), while the same number of requests uses only 10 MB on Apache for Windows NT. Scaling Apache on Unix to process larger numbers of simultaneous requests typically requires many more gigabytes of memory more than Apache on NT.

Another product which took the multi-threading approach, Netscape's web server, can scale even further than Apache can with much less memory. When it was first announced, it was able to scale on Unix much better than anything else that was available at the time, including Apache.

Like Windows NT and Apache on NT, both Lyris List Server version 3.0 and L-Soft LSMTP use this multi-threading technique. Both are capable of sending mail between 5 and 10 times faster than qmail, sending between 100,000 and 300,000 unique messages per hour. L-Soft even prices their product based on the number of send threads allowed to run simultaneously. With Lyris 3.0, we took a similar approach; a lower price limits user to an equivalently lower number of simultaneous send threads.

Thus, the second generation method is markedly faster and more scalable than the first generation. However, because multi-threading is a sophisticated technique, not all operating systems support it. The smaller Unix vendors tend to either not offer multi-threading, or what they offer is primitive and unstable. And again, for all of the advantages of the second generation approach, there are again limits and diminishing returns. The scalability of this technique definitely has a ceiling.

It has been our experience that in real world use, Windows NT is incapable of running more than 1000 simultaneous threads at a time. Each thread has its own execution context and typically points to different parts of memory. As the operating system switches between each and every currently running thread thousands of times per second, it must constantly readjust its execution context. If the memory location that a thread needs is not immediately available, a "page fault" occurs. The page fault then causes the operating system to fetch the needed memory and make it available to the thread.

This operation is relatively efficient using a small number of threads. But as the number of threads increases, the amount of CPU time dedicated to switching execution context grows. The resulting page faults increase until mailing speed drops dramatically because of operating system inefficiencies.

The page fault problem, combined with the overhead of processing large numbers of threads, limits Windows NT from simultaneously running more

than 1000 threads. Increasing threads beyond this limit causes performance to suffer dramatically, as the overhead of managing that many threads overtakes the operating system.

The same limit is found in other second generation servers. In practical tests on average NT machines, we find that Lyris version 3.0 runs best between 500 and 800 threads, and on high end machines at around 800-1000 threads. In conversation with large LSMTP customers, we have likewise been told that LSMTP will not typically go beyond 1000 threads on Windows NT. On Solaris, Lyris 3.0 can scale marginally better with higher end hardware. However, an upper limit on simultaneous threads still exists and thus performance peaks at a level below foreseeable demand.

Thus, the second generation email and web servers have used multi-threading to scale much further than the first generation. One thread per task is simply more efficient than one process per task. This one thread per task approach allows server programs which use it to scale between 5 and 10 times further than the first generation server programs.

However, there still exists a very real scalability ceiling. Beyond 1000 threads, the overhead of running additional threads is so great that adding threads beyond 1000 actually decreases overall performance in all environments. Thus, this practical 1000 thread limit imposes a very real speed ceiling: the only way to increase speed further is to use a different architecture of doing many tasks, one which would support more than 1000 simultaneous sends.

---

## Comparison Chart

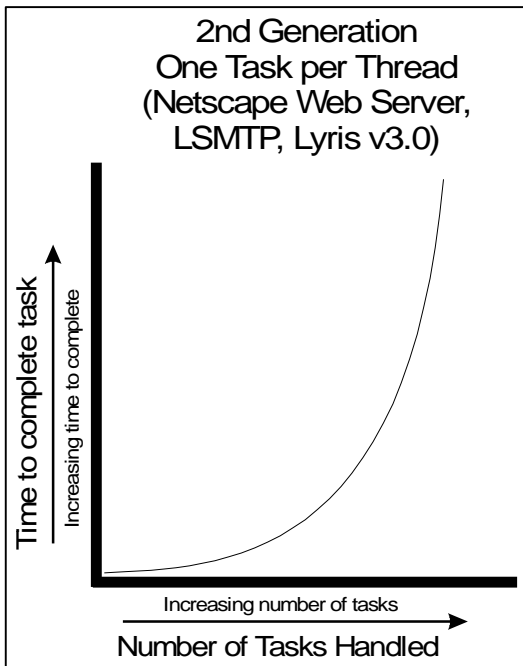


Figure 2

Figure Two illustrates the scalability of server programs which use the one thread per task technique.

At a low number of tasks, this technique performs measurably better than the one process per task technique because creating a thread to handle a particular task is faster than creating a new process.

At a medium number of tasks, this technique continues to perform fairly well: scalability increases as the number of threads increase. However, operating system overhead in managing larger numbers of threads begins to build. Performance begins to suffer because operating system multi-threading overhead requires more memory and CPU time.

As the number of tasks increases to a large number, this technique scales poorly. The operating system is spending most of the CPU time handling the swapping in and out of threads, and little time is available for the application to do its work. As one increases simultaneous threads further, performance degrades rapidly as the machine is totally consumed with multi-threading overhead. Applying additional memory or processing power increases speed at lower numbers of threads, but is not beneficial once the 1,000 thread threshold is approached.

What is the effect of adding multiple CPU's to address these limitations? While multiprocessor machines are able to distribute threads

across CPUs, as the threads increase, the size of the management algorithm offsets the benefits obtained by the distribution in the first place, and minimal performance gain is obtained. Our experience has shown that both Windows NT and Solaris are largely unsuccessful in distributing a single heavily multithreaded application across several CPUs. The application tends to dominate one CPU and only use a portion of subsequent CPUs.

# The Third Generation: Many tasks per thread, and notify when done

## Summary

In order to mitigate the delays inherent in the "wait for an answer before proceeding" approach, programmers create as many thread processes as the system can handle so that while one process or thread is waiting for an answer and doing nothing, another thread can proceed. Unfortunately, as we have seen, launching threads and processes have their own associated costs and limitations. It is not scalable beyond the point at which the operating system no longer can effectively create processes or threads.

To break through the performance ceiling established by the one task per thread second generation, a completely different design architecture needed to be created: the third generation of servers.

---

## How the third generation works

What if it were possible to have one thread handle many tasks at one time? The operating system overhead created by juggling hundreds of separate threads would be eliminated. On top of that, what if the program was able to multitask during the wait for tasks to complete and be notified when they are completed?

In standard programming, the program waits and does nothing while the operating system completes the call function. Instead of having one thread perform one task and wait, it is possible to design a TCP/IP application so that it performs operating system calls which do not require an answer back

immediately. In third generation design, the operating system accepts the command, lets the application proceed and notifies the application at some later time when the action has completed successfully.

In concrete terms, this architecture means that a single thread is capable of handling several thousand simultaneous tasks, with performance degradation being strictly linear as the tasks increase. With the first and second generation methodologies, the techniques used to support simultaneous tasks have a high overhead in memory and CPU time, and that overhead increases as the number of simultaneous tasks increase.

The impressive result is that the third generation servers do not suffer from the same kind of performance degradation curve shown by the other generations. As simultaneous tasks are added, the time to complete that task increases linearly. The only foreseeable limits to the number of simultaneous tasks that can be assigned to a third generation engine are 1) the inability of the CPU to do some tasks in an acceptable amount of time if too many tasks are generated, and 2) the memory necessary to hold all the tasks currently being handled.

However, since all that is being stored is the task information, and no per process or per thread overhead is associated with the task, substantially more tasks can be stored in memory, which vastly improves performance and raises limits significantly.

When using a third generation server program, users will enjoy much greater scalability when adding additional CPUs. Because one thread is accomplishing so much, it is possible to attach one thread directly to a specific CPU so that the operating system does not have to determine how to distribute separate threaded tasks to separate CPUs. This easily defined division of labor means that on a 4 CPU machine, four task handling threads can be created, and each one assigned directly to a CPU. The full capability of the four CPU machine is optimally used and no resources are wasted on allocation and task monitoring, delivering much higher levels of speed through distributed processing. This optimized scalability is in contrast to the first and second generation server programs, for which adding CPUs beyond certain limits produces significantly diminishing returns.

Despite the clear advantages of a third generation architecture, few operating systems have the capabilities to do this type of multi-task and notify, and fewer document it so its power can be unleashed. Only a handful of modern Unix operating systems and versions of Windows NT from 4.0 and later are able to support third generation web servers; none of them are email servers.

Microsoft specifically added this capability to Windows NT 4.0 in order to break through fundamental performance barriers they were experiencing with the Microsoft IIS web server. Product reviews of IIS vs. Netscape web server at the time show the advantage of third vs. second generation architectures. The Netscape web server performs as well as IIS at low concurrent tasks loads. As the number of concurrent tasks rise, Netscape web server performs increasingly worse with page load times taking increasingly longer, while the IIS page load time increases linearly with a gentle slope as concurrent tasks increase. For a time, this capability was undocumented. However, Microsoft eventually documented these operating system techniques after sharp criticism.

Linux also has this third generation capability, but it is not documented as a standard TCP/IP function and is only to be found by digging through the Linux kernel source code. Only Sun Solaris Unix fully supports and

documents its third generation architecture, and it is not a coincidence that Solaris has enjoyed Internet success and the very high network application throughputs that specifically take advantage of these capabilities.<sup>3</sup>

As revolutionary as this third generation server design is, it does have a downside. The programming needed to write a functioning third generation server program is much more complex than that needed for first or second generation server programs.

The complexity of its programming is one reason why, in the standard TCP/IP textbook "TCP/IP Illustrated", the author Richard Stevens argues that it is rarely cost-effective to use this technique. Since few applications need the speed that it can deliver and because the complexity for the developer is much increased, the cost and risk for the developing vendor corporation is subsequently much higher.

To date, only the highest performance web servers and a few high-performance ftp servers have implemented third generation architectures. It is likely that the proprietary web servers running sites such as Yahoo implement such technology, but there is no way to tell since they do not discuss such proprietary performance scaling secrets.

As of the writing of this paper, ListManager Extreme, the new email engine

developed and included in Lyris ListManager and Lyris MarketDirect, is the only email delivery product available which uses a third generation architecture. This advanced program design leads us to expect that, as we decrease the CPU needs of our application through optimization, and as CPU speeds increase, our email sending capabilities will increase linearly over time. By implementing this architecture, we can execute more simultaneous tasks without increasing operating system overhead.

Switching Lyris Technologies' entire email infrastructure from a straightforward and well-performing second generation engine has meant substantial cost and risk to the company. Several programmer decades of background effort and over 18 months of dedicated work went into this development just to get the first version out.

However, we are confident that this effort was justified, because now we have an architecture and a platform which is poised to scale for the future.

From our preliminary tests, the connection from a ListManager Extreme server to the Internet may be the most significant performance bottleneck. But as higher speed Internet connections become more affordable, we expect this limit to be increasingly insignificant. For the many companies who have already invested in high-speed connections, this limitation is already not an issue.

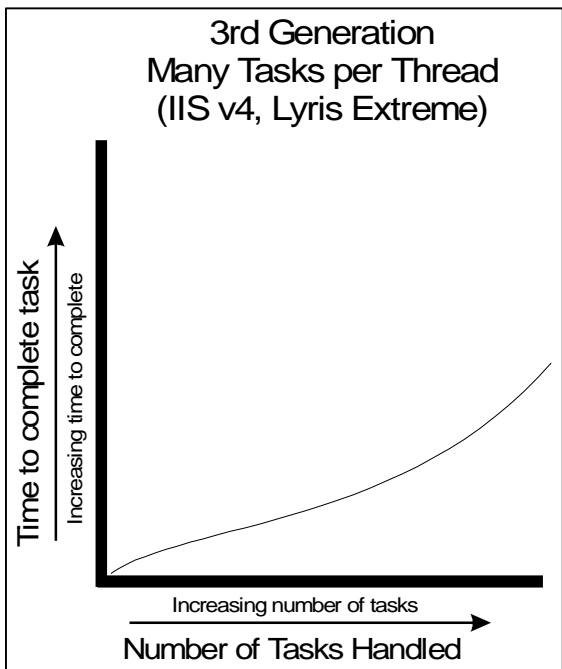


Figure 3

---

## Comparison Graph

As shown in Figure Three, a third generation server program performs well for a low number of concurrent tasks. Its performance will be similar to a second generation server program.

For a medium number of concurrent tasks, a third generation server program will begin to perform better than a second generation server program. And at

the upper levels, the key breakthrough is demonstrated; the rate of overhead remains stable as tasks increase.

As numbers of concurrent tasks rise, a third generation server program is the only choice because the other architectures are simply not capable of scaling to this level of usage. As additional speed is needed, additional CPUs or an increase the CPU clock speed would increase performance in a corresponding amount in a linear fashion.

Our investment in time and resources to produce the third generation architecture is realizing substantial benefits. We have confirmed that Lyris ListManager Extreme is able to open and simultaneously process 5,000 open connections, and this feat was accomplished using a very modest configuration, a Pentium II 350 using 256mg of RAM.

Over the next few months, we will be working closely with our customers to identify and remove performance constraints while optimizing speed. We anticipate that we will achieve the ability for ListManager Extreme to send one million unique messages per hour by the end of the year 2000 and possibly to break that mark by a substantial margin in the following year.

# Notes

## **Apache vs. sendmail on Unix**

1. There is one subtle but important difference between Apache and sendmail on Unix: when an email message is sent using sendmail, that sendmail process exits. In other words, a brand-new sendmail process is created for each email message that needs to be sent, and does not exit. With Apache, after the web page has been sent to the web browser, the Apache process waits for the next request. This is a significant performance improvement for Apache, one that has helped Apache remain competitive against second generation products in many cases. This optimization benefits Apache because creating a new process is a time-consuming step, and is an optimization that would have benefited sendmail. However, because of sendmail's architecture, it is unclear whether this "process caching" technique could be implemented. In order to try to work around the overhead of creating separate processes for every single task, some versions of Unix, notably Compaq's Digital Unix for DEC Alpha computers, have their own process creation caching techniques, which works for many repetitive process creation tasks and helps improve the performance of first generation servers.

## **Windows NT lack of forking and Apache, sendmail**

2. This lack of forking on Windows NT is the reason why both Apache and sendmail were so extremely late in being moved to Windows NT: Apache needed to be rewritten for multi-threading, and sendmail needed to be re-designed to work without the fork() call. To this day, an open source version of sendmail is only available on Unix, not on Windows NT, so it is not possible to review the Windows NT design for sendmail.

## **Other third generation servers: NaviServer and AOLServer**

3. This third generation technique was also part of a product revision made by a forward thinking web server company named NaviServer. AOL purchased NaviServer and renamed the product AOLServer. For a short time, this capability was in AOLServer, which was a freely available product. However, when AOL purchased Netscape Corp., the AOLServer project had

its key features removed, including this one-thread-handles-many-tasks feature (see the [www.aolserver.com](http://www.aolserver.com) manual for more information).

## **A note on comparable-measurement standards**

It is important to note that when making comparisons of the volume of sends that across the email industry, the benchmark number used to measure email servers is the number of unique email messages a server can send in one hour. The word “unique” is important here, because a number of email server vendors try void the comparison by quoting identical message send speeds. This practice is misleading, because it is easy to optimize this process to inflate one’s statistics. For example, one email message to 10,000 people is only one task; one email message, but with 10,000 recipients. In the industry, this is seen as one message delivered, but in an effort to artificially increase their numbers, some email vendors count this as 10,000 messages. Since the vast majority of email messages on the Internet are unique messages due to mail merging or customization, this measuring technique is artificial and should not be trusted.