Sources of Scalability Problems

There are a number of resource problems related to scaling up the system. For solutions see Scalability Solutions.

Large Number Of Objects

We usually get into scaling problems when the number of objects gets larger. Clearly resource usage of all types is stressed as the number of objects grow.

Continuous Failures Makes An Infinite Event Stream

During large network testing there is never time for the system recover. We are in a continual state of stress.

Lots of High Priority Work

For example, rerouting is a high priority activity. If there is a large amount of rerouting work that can't be shed or squelched then resources will be continuously consumed to support the high priority work.

Larger Data Streams

As the sizes of media assets grow larger then the system load increases.

As the number of sources of requests increase the system load increases.

Feature Creep

Holes in the system are uncovered as more features are added than the system was ever designed for.

Increases in Clients

Threads are setup for clients so events can be pushed to them. Memory is used for client queues. Network bandwidth is used for communication. Per client data are maintained for each client.

Less Than Good Design Decisions

There are numerous design problems that can contribute to scaling problems.

- Design that didn't handle large numbers of objects.

- Lack of end-to-end application level flow control.
- Application level retries that cause the reallocation and sending of messages.
- Overhead in the memory library.
- Publishes that are not really reliable.
- Extravagant memory usage by particular data structures.
- Messaging protocols that don't handle all failure scenarios.
- Using a hard disk for storage.
- Not using block replication for disk syncing.
- Application level protocols that could be better.
- Under powered CPU for increased feature load.
- OS that doesn't support process architecture.
- Lack of hardware support for operations that are sensitive to even a single message drop.
- Funky network problems like ARPs that get dropped as network loads increase.

Specific Scalability Problems

Assumptions Are Invalidated

Large numbers invalidate most of the assumptions you have been making about how much memory you are using, how long something takes, what are reasonable timeouts, how much you can consume at a time, how likely failures are to happen, the latency at various points in the system, how big your queues need to be, etc.

Out of Memory

The base line memory usage increases and the spike memory usage increases. These can cause OOM.

CPU Starvation

With more objects operations take a longer because they have to operate over more objects. This makes less CPU available which may starve other operations. Starvation in one place can propagate to other places.

There may not be enough CPU to take care of the required work that needs to be done. This can because the base line amount of work is high or certain scenarios cause a lot of high priority work to be done.

Raw Resource Usage Increases

More objects take more memory. If someone wanted to support 10 million simultaneous objects you may not be able to because you simply would not have enough memory.

Implied Resource Usage Increased

Most features will have a lot of additional resources used for every resource used in the "raw" resource usage.

If you are storing an object in two different lists then increasing the number of objects increases memory usage by two times.

Queue sizes may also need to adjust upwards. The amount of disk needed increases. The time to replicate the data to the secondary increases. The time to load the data into an application increases. The CPU usage increases to handle all this. The boot time increases.

Latency Increases

The latency you experience may be totally different as scale increases. CPU starvation is the major cause here.

Task Priorities are Shown to be Wrong

Task priority schemes that may have worked under a lite load can cause problems under heavier load.

In particular, when there is a poor flow control mechanism, a high priority task feeding work to a lower priority task causes drops and spike memory usage because the lower priority task will get very little chance to run.

Queue Sizes Not Big Enough

A larger number of objects imply more simultaneous operations can be made which means queues sizes will probably need to increase.

Boot Times are Longer

The more objects the longer it takes to load them into applications from the disk.

Sync Times are Longer

The more objects the longer it takes applications to sync data between each other.

Not as Much Testing in Large Configurations

Because the test setups are so expensive the actual time spent testing large configurations is small. You will not have access to large systems during development so it's likely your disgn will not scale at first.

Operations Take Longer

If an operation is applied to every object then it will take longer than it used to as more objects are added.

Tables are larger so maybe a search that was fast enough with one amount of data now take a lot longer.

More Random Failures

You may not see certain failures in normal operation. Under scale replies will drop, ARP requests will drop, the file system may show certain errors, messages may drop, replies may drop, etc.

Bigger Windows for Failure

Scale means there is more opportunity for failure to occur because everything takes longer. A protocol for exchanging data may occur quickly with a small data set, which means it has a smaller change for seeing a reboot or timeout, but with larger scales the windows increase which means new problems may be seen for the first time.

Timeouts Don't Scale

Any timeouts that worked for smaller data sets won't apply as data sets get larger. Add the CPU starvation issue, where your code may not even get a chance to run, and timeouts are further invalidated.

Retries Don't Scale

There is no way to for an application to pick the number of retries before failure is declared because they do not have enough information on which to base the decision. Is 4 retries with a second retry any good? Why not 20 retries?

Priority Inheritance

Locks that are at large scope are held for longer which makes for a better chance for seeing priority inheritance.

Events Multiply

See Continuous Failures Makes An Infinite Event Stream.

Consumption Patterns Break

At one scale you could take all of the data from producer. But at another scale you run out of queue space/memory.

An example of this is a poller that used to poll all data sources from a remote queue before moving on to the next queue. This worked well when there was a small number of items on the queue. But then a feature change expanded the number of items on the remote queue and the poller caused a node to run out of memory. The poller was changed to take a few batches at a time.

Watch Dog Timeouts

A 100% CPU condition can cause a watchdog timeout. This may rarely happen on smaller scale systems, but poor design at larger scales can make it happen.

Slow Memory Leaks Become Fast Leaks

A memory leak that went unnotices in a smaller scale may become significant at larger scales.

Missed Locks Become Noticed

If a lock should be in place, but is not, can go unnoticed in smaller scale system because a thread may never give up the processor right before the instruction that will cause the problem.

In larger scale systems there will be more preemption which means there is more of a chance of seeing simultaneous access to data by different threads.

Chance of Dead Lock Increases

Different scheduling patterns exercises code along different paths, so have a greater chance of seeing deadlock.

For example, when htfs (the file system), doesn't get a chance to run because of high CPU usage, it somehow breaks in a way that it takes 100% CPU and never runs again.

Time Syncronization Suffers

Time syncronization is not a very high priority, so as CPU and network resources become less available clocks on different nodes will drift.

Logger Drops Data

The logger can start dropping data because its queues are too small to handle the increased load or the CPU is too busy to give time to the logger to dispatch the log data. Depending on the size and type of the queues, this can cause OOM.

Timers Don't Fire at the Right Time

A busy system will not be able to fire timers at the expected time, which can cause a cascade of delay in the rest of the system.

ARP Drops

During high CPU or network loads ARPs can drop between machines. This means that packets are sent to the wrong cards until the tables are updated, which can be never.

File Descriptor Limits

There is usually a fixed limit to the number of file descriptors available on a box.

Design must limit the max number to be under the limit. If socked descriptors are taken out of the file descriptor pool, then a design with a large number of connections (ftp, com, booting, clients, etc) will cause problems.

With scale it's possible to have a spike of the number of descriptors.

Descriptor leaks will use up the available pool as scale increases.

Socket Buffer Limits

Every socket has an allocated amount of buffer space. Large numbers of sockets can reduce the overall amount of available memory.

As scale increases drops increase because the amount of buffer space to receive messages is not enough to keep up with the load. This is also related to priority because a task may not have enough priority to read data out of the socket. On the sender side a high priority task may overwhelm with messages a task with lower priority.

Boot Image Serving Limit

The number of booting cards a node can serve is limited to X at a time. The ftp server infrastructure must limit the number of cards it servers or the node gets CPU starved.

Out of Order Messaging

Under stress your Com system may start delivering messages out-of-order which can cause problems for non-idempotent operations.

For example, an out-of-order message caused database replication to unexpectedly stop.

Protocol Weakness

Unless application protocols are created very carefully increased scale brings out a lot of problems. See Ps Cursor Violation OfIdempotency And Statelessness for an example.

Connection Limits

A central server of some kind with ten clients may perform adequately but with a thousand clients it might fail to meet response time requirements. In this case, the average response time probably scales linearly with the number of clients, we say it has a complexity of O(N) ("order N") but there are problems with other complexities. E.g. if we want N nodes in a network to be able to communicate with each other, we could connect each one to a central exchange, requiring O(N) wires or we could provide a direct connection between each pair, requiring O(N^2) wires (the exact number or formula is not usually so important as the highest power of N involved).

Tiered Architectures

This is a good a summary so I'll just refer to it here:
http://jroller.com/page/gnirpaz?entry=tiered_based_architecture_is_broken

Tiered based architecture was never meant to enable building low latency, high throughput applications. The fundamental problem with tiered architecture is that it was created to solve yesterday‘s problem. In the transition from the client-server era to internet time, it was the perfect solution for scalability.

The problem domain was how to scale applications to support hundred thousands of users. The solution for this problem was the n-tier architecture that we know today. The scalability dimension that was chosen was the presentation layer through load balancers. Indeed it actually solved this problem.

However, now days, the problem has evolved. These days in many industries the problem is not about just scaling up the user experience, it is also a question of data volumes. For example, in the financial industry, the move from doing mostly manual trading, to automatic trading systems increased the data volume almost exponentially.

So, what is the solution for this? The answer is simple Google. You cannot imagine Google architecture to be centralized around database servers.

State Management

We can use the traffic jam analogy. When there is a traffic jam in the same place everyday, what can the municipality do to reduce the contention? They will create more traffic lanes to enable

the traffic flow in parallel (actually they will only do this if elections are coming). This will parallelize of the traffic flow. However, there is one more optimization that can be done; if all those roads are bumpy country roads, even with the fastest cars, we will not be able to exploit the cars‘ turbo engines. In this case, we need to improve the quality of the road.

In server technology, the first step is to partition the data to reduce the database contention (more lanes). The next step is to bring the data closer to the application. This is done by moving the data from physical disks to the most efficient media which is memory (efficient roads). This solves the data contention problem.

Processing

The second problem with the tiered approach is the processing inefficiency. Utilization of processing cycles is very low. If you check which code is actually running, you‘ll be amazed to find that most of the time servers do data conversion and remote calls (70%), only small amount of the time is used to actual business processing. This doesn‘t say that logical separation of tiers isn‘t valuable ? it is; the physical separation is problematic (to say the least?) The processing should local, in same address space and use object references instead of copies. In this model development productivity is higher, since the programming model is simpler.