Engineering Realities of Building a Working Peer-to-Peer System

Michael B. Jones and John Dunagan

June 2004

Technical Report
MSR-TR-2004-54

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Engineering Realities of Building a Working Peer-to-Peer System

Michael B. Jones and John Dunagan

Microsoft Research, Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
USA

Abstract

The Herald project at Microsoft Research has built working implementations of several scalable peer-to-peer algorithms as part of our work on a scalable, fault-tolerant event notification system. Our goal has been to construct and validate implementations that will work on real networks at scale – not just to simulate such systems and reason about what might be buildable – but to actually build and run them.

This paper reports on our experiences building a working peer-to-peer system, relating lessons learned that will potentially be useful to others contemplating similar endeavors. Our experiences and recommendations fall broadly into two categories: (1) We strongly recommend that the system be developed such that it can be run in both simulated and real network environments. (2) We observed that message transport issues such as protocols used, overlay network characteristics, multi-hop routing, layering, and race conditions introduce significant non-local complexities and interactions of which applications must be aware to function correctly. We built real peer-to-peer systems that scale to hundreds of thousands of nodes in a network simulator while also running on real networks, including those with WAN characteristics. We believe the lessons learned along the way will be useful and interesting to others trying to build real working scalable systems.

1. Introduction

Numerous projects exist today that propose scalable peer-to-peer solutions to real-world problems. Yet evaluating these solutions poses a dilemma. Real-world experiments involving hundreds of thousands of nodes or more – enough to validate some of the desired scaling properties of the system being built – are typically infeasible because of the resources required. And yet simulation experiments, which can be run at these scales, by necessity sacrifice some of the real-world details that actual deployed systems must handle correctly. And while real-world experiments with hundreds of nodes are increasingly feasible by employing testbeds such as Netbed/Emulab [White et al. 02] and Planetlab [Bavier et al 04], such small-scale deployments can not verify that the algorithms employed will scale to much larger numbers of participants.

Facing this dilemma, our conclusion was that the best way to convince ourselves that the systems we built would work both on real networks and at scale was to test them in two different environments: on real networks of machines at modest scale and in simulated networks at much larger scale. This approach was successful for us, but not without its trials and pitfalls.

Finally, before we begin reporting our experiences, we recognize that several other projects, such as Macedon [Rodriguez et al. 04], have also employed techniques similar to those described here, such as enabling peer-to-peer systems to be run in both simulation and live environments. This paper is distinguished from others, however, in that we are focusing on the techniques employed in building the system and the engineering realities faced while doing so, rather than the results obtained by the system itself.

We believe that many of the practical lessons learned and “war stories” from building working peer-to-peer systems have not been reported to the community so that others can benefit from them. This paper aims to fill this gap.

2. Develop for Both Simulation and Reality

2.1 Single Code Base

One decision we strongly stand behind is that we used a common code base for both simulation and live experiments. Most software components were compiled into libraries that could be run in either environment. For instance, the SkipNet [Harvey et al. 03] overlay network and the Overlook [Theimer & Jones 02] scalable name service used the same code in both environments. The message transport interface was abstracted, with transport providers implemented for TCP and a network simulator derived from the one built for the Pastry project [Rowstron & Druschel 01]. Of course, some environment-specific code was just different. For instance, all nodes were created in a single program in the simulator environment, whereas each ran in its own process in the live environment.

It might be tempting to think that first you simulate, while coding and debugging your algorithms, then you run them on real systems with no reason to go back to simulation once you're running "in the wild". But in our experience, this is definitely not true. Problems or insights discovered while running live caused us to redesign algorithms on several occasions. As discussed later, it's far easier to debug in the simulation environment. Thus, as our systems evolved, we greatly benefited from being able to go back and forth between simulations and live runs. The lesson: Don't throw your simulation environment away once you're (also) able run in a real network environment. As your system evolves (which they always do!), having it in your arsenal of tools will keep paying off.

2.2 Single-Threaded Code

While we believe that using a single code base was the right decision, it was not without significant pain. Being able to simulate hundreds of thousands of nodes required minimizing the physical memory usage per node. In particular, there was no lightweight threads package available for our implementation environment (C#) that would have kept thread stacks small enough to be able to afford one per node. Thus, we were forced into a single-threaded event-driven programming style with hand-written continuations.

This programming style can certainly work, but it results in quite obfuscated code, where, for instance, the code to send a request message and the code to process the reply for that request are in different procedures, with state passed explicitly between them in continuation structures. Lightweight threads a la Capriccio [von Behren et al. 03] and/or language support for continuations would have certainly made our lives easier and the code more readable. We encourage others to use such support if it exists for your chosen programming environment.

2.3 Message-Level versus Packet-Level Simulation

Our project used a message-level network simulator derived from the one built for the Pastry project. Message-level simulations are certainly not as accurate as packet-level network simulations, such as the NS-based simulations employed by the Macedon project, since message-level simulation fails to capture all the congestion and queuing behavior that packet-level simulation includes.

Nonetheless, each type of simulation has its strengths and weaknesses. Macedon's experience was that they could only simulate up to a few hundred nodes with packet-level simulation – a typical result. This is still quite useful for initial debugging. However, this type of simulation could not be used to verify scaling properties of algorithms.

The message-level simulator we used could easily perform simulations with hundreds of thousands of nodes. Thus, we could observe behavior of the systems running at decent scale. Of course, one should ask: "But are message-level simulations accurate enough?" Our answer is "yes, with caveats". Provided the experiments we were running didn't cause network overload in which congestion and queuing came into play, our simulation results typically closely matched results obtained on our cluster of 40 machines on which we typically ran 400 node processes.

Our recommendation: If system scaling or simulation execution speed are important to you, use a message-level simulator. Better yet, if you have access to both kinds of simulators, test your code in both!

2.4 Reproducible Execution

Possibly the most valuable aspect of the simulation environment is that runs can be deterministic and reproducible – something not true on a live network. We allowed seed values to be specified on the command line so that different runs could employ different sets of random number choices while still being reproducible. Reproducibility has obvious advantages when debugging your algorithms.

2.5 Simulated Time versus Real Time

Another advantage of debugging in a message-level simulator is that some experiments can run orders of magnitude quicker than they would on a real network, particularly if a WAN is being employed. This is because WAN message delivery times are measured in tens or hundreds of milliseconds, whereas simulator messages can be delivered in fractions of a microsecond. Of course, if the experiment is highly parallel the simulator runs may be slower, but for any serial phases, it's nearly always faster.

Another significant factor in how long experiments take to run is the set-up and management overhead of running on a cluster or true distributed system. On our cluster, in addition to the usual the edit-compile-debug cycle, additional steps were needed to (1) reserve nodes on the cluster, (2) copy the compiled binaries to all the cluster nodes, (3) start the nodes in a staggered fashion, (4) wait for all nodes to exit, and (5) copy results from all nodes to a results repository. Don't underestimate the time required for mundane but essential tasks when running live experiments.

2.6 Debuggers versus Log Files

Because simulated runs occur entirely within a single process, they can be debugged using standard debuggers and techniques. In our environment, they are just single-threaded deterministic program executions, making it easy to drill down through abstraction layers and find out what went wrong when something did. But remember…

2.6.1 Even though you've debugged it on the simulator, it still may not work in the wild.

For all the reasons that network simulations do not faithfully mimic real networks, the same code run in the two environments can and will behave differently. A handful of differences we had to debug in the live system included:

- different execution orders,
- different ranges of message delivery times,
- unexpected timeouts occurring,
- dropped messages due to network saturation.

All of which brings us to the point that…

2.6.2 You *will* have to debug with log files.

Unless your distributed system debugging environment is head-and-shoulders above ours, you will find yourself debugging distributed algorithms by doing post-mortem analysis of node log file outputs from failed runs. It's not pretty or fun, but here are some principles we applied:

- Debug with log files as rarely as possible. (It's *much* more efficient to debug using the simulator!)
- Log files should contain enough state to verify key program invariants.
- Log files should contain enough state to enable relevant distributed execution paths to be retraced.
- Always log. You never know when a bug will manifest itself and you may not be able to reproduce it later.

2.7 Closing Remarks on Simulation versus Reality

In our experience, having the simulator was invaluable but it's not the same as reality. Do the differences matter? It depends. For instance, in our simulator, the first message exchange between two nodes takes the same time as the second such exchange. On the live network, the first exchange is slower because of the TCP connection setup handshake that occurs before data is sent. The simulated time is very close to the time for the second (and subsequent) exchanges. Is this a problem? It depends upon how you're using the results. The main lesson here is to take advantage of the strengths of both environments, while being fully aware of their weaknesses.

3. Face the Transport Mess

Sending a message from one node to another is a conceptually simple operation with a potentially large amount of complexity and factors to be aware of under the covers, particularly when peer-to-peer overlay networks are employed. This section explores some of these issues we faced in our work.

3.1 Choice and Consequences of Choosing TCP

We chose TCP for our node-to-node transport because it handles message delivery retries for us, hence the layers above it wouldn't have to. Plus, we wanted the "friendly" congestion backoff TCP provides, letting us be good network citizens. However, this choice came at a price.

A typical TCP message can be delivered in milliseconds. But, in the face of failure, TCP keeps trying to deliver a message for quite a while – in our implementation, over half a minute. Thus, there are many orders of magnitude difference between successful and unsuccessful message times. Furthermore, in the presence of congestion, even successful communication can take three orders of magnitude longer than typical times.

The high variance and long timeouts that come with using TCP effectively meant that while failures are rare, when they do occur, higher-level recovery and re-routing can take minutes to accomplish.

Others have made choices other than TCP, including the use of UDP with application-level retries and hybrid transport protocols [Dabek at al. 04], with the combination tuned to the characteristics of a particular application. While not covering these choices per se, we hope that this description of our experiences using TCP for overlay transport, both good and bad, will help inform other's choices of transport protocols for their systems.

One meta-point to consider when comparing transport possibilities is that each choice, such as TCP, UDP, and others, is actually a choice among a complex set of many inter-related properties, including retry semantics, congestion behavior, handling of out-of-order packets, support for messages longer than a single packet, acknowledgment semantics, timeout values, etc. If your application cares about these properties, you'll either need to choose a transport that provides them in an acceptable manner or implement them yourself in some fashion. Our experiences have led us to conclude that there's not a one-size-fits-all transport choice that is best for all peer-to-peer applications.

3.2 Layering can get you in trouble.

Our peer-to-peer applications are built using software layers that themselves use one or more layers below them. For instance, one application uses application-level multicast trees similar to Scribe trees [Rowstron et al. 01]. The multicast trees use the SkipNet overlay network. The overlay uses a transport – either the simulator or TCP. Each layer provides useful services to its clients. But each also implicitly inherits the characteristics of and problems with the layers used to construct it.

Consider timeouts. As already discussed, TCP takes on the order of a minute before declaring a connection failed. This means that SkipNet, which is built on it, must use failure detection timeouts longer than the TCP timeout, or else handle the complexity of declaring a failure when the communication may still succeed. In practice, this meant that SkipNet routing table repairs might not occur until several minutes after a node or link failure – once the ping message detecting the failure had timed out.

By default, our SkipNet implementation retries message deliveries three times (trying different routes upon failure). These retries will be most effective only if routing table repairs have occurred between them – the time for which is dependent upon both the ping and repair message timeouts.

Next, the clients of SkipNet may have their own timeout values. For instance, the user-level multicast code may need to wait for the SkipNet routing table to be repaired before building a new reverse-path forwarding tree when nodes fail or join the overlay. By now, in this example, conservative timeout values may be several minutes long.

Each layer's timeout values depend in potentially non-obvious ways on the timeout values of the layers they use. We call this problem "cascading timeouts". It is real, and if you are unaware of it, it can bite you in numerous ways.

Our best advice is that each layer needs to understand the strategy of and have visibility into the state of the layers beneath. Information hiding doesn't serve us well here – state transparency does. However, state transparency is not a panacea, since it only provides visibility to state on your own node, whereas in a peer-to-peer system, a node's behavior will often depend upon the states of other nodes it communicates with. No easy answers here…

3.3 Issues with Multi-Hop Delivery

Some overlay networks, such as SkipNet and Pastry, perform multi-hop delivery of messages between application nodes, meaning that each node a message is routed to forwards the message on until it reaches its destination node. While a convenience to applications and yielding more efficient delivery (0.6 times the cost of iterative delivery according to [Dabek at al. 04]), this decision comes with its own set of issues.

3.3.1 End-to-End Reliability

TCP was designed for end-to-end reliability. If you get an ACK, you know that the message was delivered between TCP endpoints. Multi-hop delivery loses this property since message delivery can fail at intermediate nodes without the sender being notified.

3.3.2 Timeouts

Multi-hop delivery also makes the setting of timeouts even harder than described previously. For instance, setting a timeout for end-to-end message delivery requires a conservative upper bound on the number of hops (as well as a conservative bound on the per-hop delivery time).

3.3.3 Congestion Control

Hop-by-hop congestion control interacts poorly with end-to-end congestion control and end-to-end latency. For instance, consider a situation in which a stream of data is being sent using the overlay between two nodes via an intermediate node. In this case, the intermediate node may have to buffer an unbounded amount of traffic if the first link has higher bandwidth than the second because there is no push-back mechanism. TCP's hop-by-hop congestion control does not achieve end-to-end congestion control when multi-hop delivery is employed.

3.3.4 Advantages of Multi-Hop Delivery

While it would be tempting to conclude from the above issues with multi-hop delivery that iterative delivery should be preferred, the reality is more complex than that. For one thing, even if the overlay network uses iterative delivery, the next layer up may be performing multi-hop delivery anyway. For instance, overlay multicast is implemented as multi-hop delivery with fan-out.

There are also engineering advantages of multi-hop delivery over iterative, besides the performance advantage already cited. One is that each node's overlay communication occurs only with its routing table neighbors – a relatively small set, allowing connections for all active routes to be cached and reused. Another locality-related advantage is that communication failures are detected by routing table neighbors, facilitating quick local overlay repair actions. Once again, the correct choice probably depends upon your particular application characteristics.

3.4 Embrace Race Conditions and Uncertainty

Timeouts are often used in traditional communicating systems as drop-dead points beyond which one might reason "if it hasn't happened by now, it surely won't happen". But, as illustrated by the discussions above, in the presence of changes to overlay routing tables and communication patterns involving multiple nodes, setting reasonable timeout values can be quite problematic. As a real example, while developing one application we found ourselves increasing the timeout associated with creating a distributed data structure to 13 minutes! (And it still wasn't long enough in certain pathological cases.) This led us to conclude that there had to be a better way.

This "better way" involved accepting that you can't predict how long a distributed operation might take to accomplish in the presence of failures. In place of "wait for it to be done, then do the next thing" style communication patterns, we instead tried, wherever possible, to employ communication styles that would make progress towards our overall goals, even in the presence of uncertainty about the global system state.

For instance, an idempotent operation such as "Store the value x at the name /foo/bar/x-value, creating the directories /foo and /foo/bar if not found" can succeed whether or not previous distributed directory creation operations for "/foo" and "/foo/bar" had completed. Such operations admit the possibility of races between distributed operations and temporarily inconsistent state.

Our experience is that it is better to accept that races will occur and to use algorithms that make forward progress in spite of them, than to have to wait long periods of time for distributed operations to complete and still possibly not know whether they were fully successful or not.

3.5 Using Modelnet for WAN Emulation

We used Modelnet [Vahdat et al. 02] to emulate WAN network delivery characteristics on a LAN. This allowed us to validate our code both on the LAN itself and on the emulated WAN which has very different timing characteristics. This gave us further confidence that our code will operate correctly in the wide area.

Of course, the WAN emulation accuracy is still subject to the physical limits of the LAN being used to emulate it. At one point, our experiments began to fail due to dropped messages and we discovered that we were trying to send more than 100Mbits of traffic through a 100MBit link to the Modelnet router machine. This was fixed by changing the link to a 1Gbit link. But diagnosing this problem wasn't easy. It required looking at packet traces plus back-of-envelope calculations of total bandwidth based on sampling of packet sizes and rates to determine that we were overloading the link.

3.6 Data Encoding and Compression

The C# runtime libraries provide routines making it convenient to marshal native data structures into XML for transport, and to unmarshal the XML back into native data structures upon receipt. This capability made it much easier to develop peer-to-peer applications passing non-trivial data structures between themselves.

However, while expressive, XML is not space-efficient. We found that simple overlay messages, when encoded in XML, typically required about 2000 bytes of space – more than an Ethernet packet in size. Applying compression reduced this to fewer than 600 bytes. This reduced our network bandwidth requirements considerably – a fine tradeoff given that our applications were not CPU-bound.

3.7 In the real world, stuff happens.

In a real network, you need to expect the unexpected. For instance, in our cluster of 40 machines as originally configured, packet traces showed us that some packets were taking nine seconds to be delivered! They weren't being dropped and retransmitted – they were spending nine seconds somewhere in the network. We believe that this was caused by IP address collisions on the network and that the packet was being held by the Ethernet switch until the collisions were resolved. Without going into why this was the case, we spent quite a bit of time trying to figure out whether our code was causing these occasional nine-second message delivery times or whether it was in layers below us. Once again, the lesson is that, at least in terms of performance, you're never isolated from the behavior of the layers below you.

4. Conclusions

Building real, working peer-to-peer systems is a manageable engineering task using known techniques but it is far from trivial, and requires large amounts of testing before code that "should obviously work" actually does, when run on a real distributed system.

Our first conclusion is that the best decision we made was to build both for the simulation environment and a real network environment. We strongly encourage others to follow this approach. That being said, don't trust your simulation results too far. Code that works in a simulator often doesn't work on a real network and may produce different results. Similarly, while using Modelnet to emulate a WAN network helped make our systems more robust, we fully expect that running them on a real WAN environment such as Planetlab could still cause them to encounter corner cases not previously exercised.

Next, while overlay networks appear to provide a clean and useful abstraction on top of existing networks, our experience using one for real has starkly pointed out the problems with layering. Unless each layer has visibility into the state and algorithms of the layers beneath it, seemingly bizarre and inexplicable behaviors can result. Transparency between layers is a must, but is also not a panacea.

Race conditions and uncertainty are unavoidable in real distributed systems. The algorithms we employed that attempted to make forward progress in the face of these realities worked far better than those that tried to wait until known, consistent states were reached.

Finally, we'd like to close by saying that the code described in this paper is available as part of the public SkipNet release [Herald 04]. Thus, you can see for yourselves the details of these and other engineering tradeoffs we made to build several working peer-to-peer applications.

References

[Bavier et al 04] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating Systems Support for Planetary-Scale Network Services. In *Proceedings of the First Symposium on Networked System Design and Implementation* (NSDI '04), San Francisco, CA, March 2004.

[Cabrera et al. 01] Luis Felipe Cabrera, Michael B. Jones, and Marvin Theimer. Herald: Achieving a Global Event Notification Service. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems* (HotOS-VIII), Elmau, Germany, May 2001.

[Dabek at al. 04] Frank Dabek, Jinyang Li, Emil Sit, James Robertson, M. Frans Kaashoek, and Robert Morris. Designing a DHT for Low Latency and High Throughput. In *Proceedings of the First Symposium on Networked System Design and Implementation* (NSDI '04), San Francisco, CA, March 2004.

[Harvey et al. 03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: A Scalable Overlay Network with Practical Locality Properties. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems* (USITS '03), Seattle, WA, March 2003.

[Herald 04] SkipNet public code release. http://research.microsoft.com/sn/Herald/, February 2004.

[Rodriguez et al. 04] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostic, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *Proceedings of the First Symposium on Networked System Design and Implementation* (NSDI '04), San Francisco, CA, March 2004.

[Rowstron & Druschel 01] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001.

[Rowstron et al. 01] Antony Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Proceedings of the Third International Workshop on Networked Group Communication*, UCL, London, UK, November 2001.

[Stoica et al. 01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A Scalable Peer-To-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM*, San Diego, CA, pp. 149-160. August 2001.

[Theimer & Jones 02] Marvin Theimer and Michael B. Jones. Overlook: Scalable Name Service on an Overlay Network. In *Proceedings of the 22nd ICDCS*, Vienna, Austria, July 2002.

[Vahdat et al. 02] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostic, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (OSDI '02), December 2002.

[von Behren et al. 03] Rob von Behren, Jeremy Condit, Feng Zhou, George C. Necula, Eric Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of 19th ACM Symposium on Operating Systems Principles* (SOSP '03), Bolton Landing, NY, October 2003.

[White et al. 02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad Mac Newbold, Mike Hibler, Chad Barb, Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation* (OSDI '02), December 2002.