LOW-LATENCY IPTV NOTIFICATIONS WITH MINIMAL SERVER IMPACT

Gary Horton
Time Warner Cable

Abstract

IPTV platforms provide value-add with notifications such as Caller ID and EAS alerts. An efficient architecture provides timely delivery of these messaging events without unnecessarily binding resources in their absence. The technology choice to support this is influenced by message frequency, latency requirements and client population size.

Among the contexts combining those variables, low latency delivery of infrequent messages becomes more challenging for the server as client size increases. A system is described here targeting that scenario, scaling gently with increases in client size and latency constraints. The system impacts server resources only when a message is available.

INTRODUCTION

Background

With HTTP as the application protocol, notification across the Internet relies on a client-initiated request. This presents the problem of how to convey event information from a service to a client, for example an EAS service delivering a flash flood alert to an HTML5 browser.

Periodic client requests provide opportunities for the server to send the EAS alert. However, with such "short poll" solutions, the client can suffer from high latency and untimely message delivery if polling frequency is too low; or server capacity is challenged when frequency is high. In particular, the aggregate cost of setup, maintenance and teardown of TCP connections[1] increases as a function of the polling frequency and client size.

So-called COMET[2] or "long poll" solutions have been developed to reduce the TCP activity. With these approaches, a connection is established and held by the server, either until a message is available or with periodic timeouts. Streaming servers can hold the connections open persistently and deliver ongoing messages as information becomes available. These techniques not only reduce TCP setup and teardown cost, but also support the requirement for low-latency message delivery. However, while setup and teardown cost is lessened, maintenance remains: system provisioning requires concurrent connection capacity as a function of client size. This incurs cost in RAM[1] and brings other considerations[3].

The benefit of the RAM cost is a function of messaging frequency. That is, in a context with frequent message availability such as a stock exchange, the capacity is well utilized. However, that use declines with decreasing messaging frequency. Persistent connections are increasingly idle, becoming an unused commitment.

Examples calling for low-latency delivery of infrequent messages include EAS, Caller ID and location-aware advertising.

LightWeight Polling

The traffic profile combining timely delivery of infrequent messaging events is the context for this discussion. Targeting that context, "LightWeight Polling" (LWP) is a communication protocol described here, with performance test results suggesting it provides low-latency messaging while moderating many shortcomings of traditional short poll

systems. In particular, significant savings in network and CPU activity have been measured and will be discussed in detail. In short, LWP accomplishes this by opening its destination port for clients only when a message is available. Resources are used primarily for those events while terminating any TCP connection attempt immediately in their absence.

Acronyms

For purposes of this discussion, the following acronyms are used:

CPU	Central Processing Unit
EAS	Emergency Alert System
HTTP	HyperText Transfer Protocol
JVM	Java Virtual Machine
LP	Long Poll
LWP	LightWeight Poll
RAM	Random Access Memory
SP	Short Poll (traditional polling)
TCP	Transmission Control Protocol
TTL	Time To Live

ARCHITECTURAL CONSIDERATIONS

Use Cases

Testing done here is intended to simulate broadcast messages that occur infrequently but call for timely delivery.

The point-to-point class of use cases, for example Caller ID and soft remote, imply private instead of broadcast messages. This is not an intended use for LWP as it is currently implemented. The current prototype relies on a *port-per-topic* protocol, using a single destination port to host a given topic. For example, the "topic" for Caller ID is the individual phone number, and this requires N ports for N phone numbers. Supportable Caller ID client size then becomes a function of available ports[9]. While larger values of N appear to be a variation of the unused commitment problem, LWP capacity is not dedicated in the absence of messaging since ports are not opened. Given this, LWP applied with point-to-point may show benefits, but experiments remain to-date undone.

Latency requirements that are more relaxed are not examined here since this is not a particularly challenging context.

The class of use cases calling for frequent messaging, for example a stock exchange, are also not an intended application of LWP systems. As frequency of messaging increases, LWP systems increasingly resemble standard polling, and long poll systems become increasingly favorable.

System Requirements

An example set of requirements where an LWP system would be suitable include the following:

1. **one-second-latency:** The system must support latency requirements of one second or better after message availability.
2. **120s-messaging:** System capacity requirements must grow linearly or better as a function of messaging frequency with a period of 120 seconds or more.
3. **graceful-scalability:** System capacity requirements must grow in sublinear fashion relative to client population size.
4. **dynamic-capacity:** The system must release CPU and network resources when messaging work is not required.

Candidate Approaches

Given the target use cases and system requirements, there are various candidate solutions included in testing done here. The range of candidates was not intended to be exhaustive, for example omitting Web Sockets[4]. The intention was to determine if short polling with an LWP approach is a viable alternative to long poll.

In terms of the system requirements, the **120s-messaging** context is not challenging taken alone. However, low-frequency messaging undermines the **dynamic-capacity** requirement with a long poll system, which by design is unwilling to release capacity in the absence of messages. The **one-second latency** requirement is problematic with a traditional short poll system in terms of support for **graceful-scalability**[5]. This is true regardless of messaging period, due to impact on RAM and network. Increasing client sizes with a long poll system would require corresponding growth in concurrent connection capacity, failing to support **graceful-scalability**.

LIGHTWEIGHT POLLING: DETAILS

Overview

The "LightWeight Polling" mechanism described here is not sensitive to high rates of client polling, with the destination port opening only when messages appear. Without messaging, TCP connection attempts are terminated immediately. This is done with a connection reset issued from the network stack on the LWP host[6], incurring modest impact solely on that stack.

At message time, TCP communication in LWP flows as usual[7]. When messaging events are infrequent, the cost of TCP setup, maintenance and teardown are reduced relative to traditional short poll. With lessened demand on server, network capacity and CPU, these resources are available for other activities when no messages are available. LWP capacity needs are largely determined by messaging frequency.

The LWP protocol side-steps the provisioning needs of long poll solutions, where capacity for "always-open" connections must grow as a function of population size. These connections are idle a majority of the time when messages are infrequent.

The combination of "strict latency" (timely delivery) and infrequent messages is the most suitable context for an LWP system. The remainder of this discussion assumes strict-infrequent as the context.

Communication Flow

The sequence diagram in Figure 1 illustrates the typical LWP flows. The timeline is captured to the left of the Client, with system State changes to the left of the Publisher. To elaborate on the numbered steps:

1: A web Client "subscribes" for "notification" when an EAS alert occurs. The notion of EAS alerts is regarded as a "topic", and the client request is referred to as a "subscription". Port 8088 is used for the subscription, with this port remaining open for the duration of LWP execution.

The LWP service responds to the Client subscription with information on the port for polling and a TTL (Time To Live) – that is, how long any EAS alert will be cached for availability. The TTL is the maximum recommended polling interval for this particular topic.

In this example, the port to be used for polling is 8091 and the TTL is 10 seconds. If the Client polls more frequently, it is less likely to miss any messages and may receive

messages with less delay. If polling is done less frequently, the Client may miss messages.

2a-2c: The client is motivated to get any EAS alerts with little delay, polling every second. Note from the current State (No Alert, Alert, …) that each of steps 2a, 2b and 2c are executed when no alert is available.

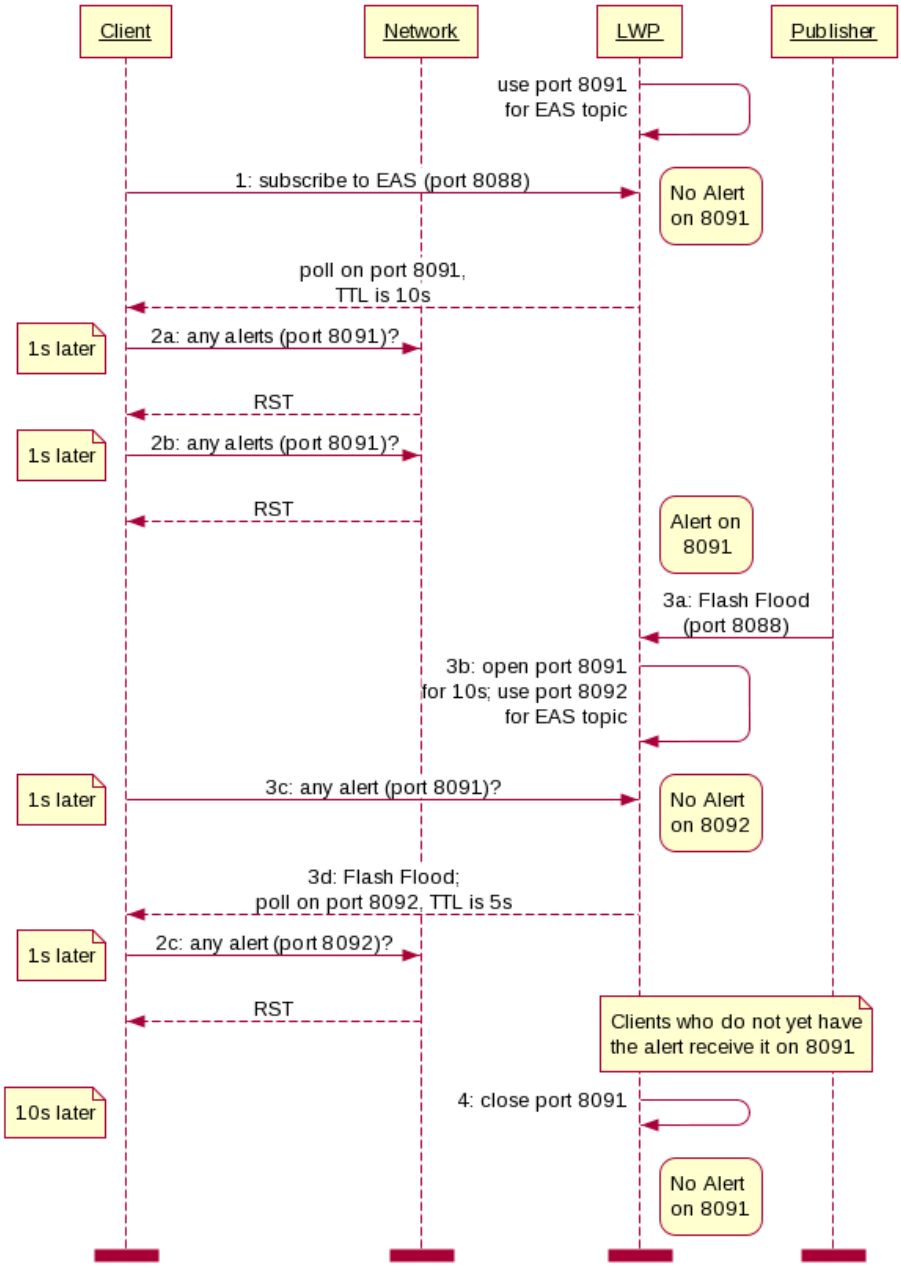


Figure 1: Typical LWP Communication Flows

The impact from one-second polling is quite low as quantified in the TEST RESULTS section. When no message is available, LWP is not aware of the polling frequency, since the Network terminates the connection attempt immediately with a TCP reset.

3a: The Publisher provides an EAS message to LWP at 3a using port 8088. LWP responds at 3b by opening port 8091 with the message and establishing port 8092 for subsequent messages. Note that port 8092 is not opened until that next message arrives. The next time any Client requests an EAS alert on 8091 (3c), the message is delivered (3d), and LWP informs that client to poll for subsequent messages on port 8092 with a TTL of 5s.

Note that the port and TTL have changed. LWP changes the port so that this first client is not repeatedly asking for the same message on port 8091, which remains open for other clients during the TTL. This would undermine the LWP goal of "no unneeded work" and result in duplicate messaging to the first client. Other clients that have not yet received the message will find it at port 8091 for the next 10 seconds. Meanwhile the first client will now poll

against 8092.

This *port-per-message* protocol requires a client to resubscribe if it has not contacted LWP within the TTL period. If a message was delivered during that TTL, the port will have changed for the EAS topic. Alternately, clients can resubscribe on port 8088 on a regular basis as insurance against changed port numbers, and a message numbering scheme can be used to deliver potentially missed messages to clients.

Note that proof-of-concept for the port-per-message protocol remains undone. Tests done here use a *port-per-topic* implementation of LWP, and this did result in duplicate messages for a given client and additional impact on LWP during the TTL period. A port-per-topic approach is sufficient for point-to-point, since this is equivalent to port-per-client and no redundant requests for the same message will happen (since the port is closed immediately after message delivery).

Finally, as illustrated in the sequence, LWP optionally changes the TTL to adaptively manage its resources as needed. TTL adjustments are also used to encourage higher polling frequencies. For example, an initial EAS alert is likely to soon be followed by more.

TEST CONSTRUCTION

Test Setup

Test results provide some insight into why traditional short poll systems are considered inefficient and how LWP addresses the problems. An LWP server prototype was constructed using the netty (Java-based) framework[8]. In the interest of reducing the variations, SP and LP server prototypes were also constructed with this framework. The SP implementation represents a traditional short poll system for purposes of testing and subsequent analysis in this discussion.

Each server was load tested in separate executions on a physical hardware machine dedicated to the tests. The machine has two Intel E6750 CPUs running at 2.66GHz. The JVM was configured at 256MB minimum and 2GB maximum heap size.

SP and LWP Test Phases

The SP and LWP systems were executed in various *test phases* as described below, and as annotated with the timeline in Figure 2 in the TEST RESULTS section.

The test phases **Q**, **P**, **S**, and **M** are, briefly:

Q: the system is quiescent
P: polling is underway
S: server is running, handling polling only
M: messages are being delivered

The test phase details are as follows:

Q: The system is in quiescent state for at least 180 seconds to establish a system baseline.

P: Polling is initiated for 300 seconds to establish a polling-only baseline. For each test, either 3K or 30K client threads are initiated in sets of 3K at a time. For the 30K population, a 6-second delay is added between each 3K set, with the intent of adding some temporal spread to the client request traffic. This results in a “bumpy” rather than a uniform distribution of client request traffic, due to staggering and resource contention delays with thread startup.

S: At the first of two “S” phases, a given type of server (SP or LWP) is started and opens port 8088 to accept publisher messages. SP opens port 8091 to accept client requests. LWP opens that port only when a message appears, and then only for the TTL period.

Polling against SP and LWP continues without any message delivery for 600 seconds to establish a no-messaging baseline.

M: Messages are delivered five times, once every 120 seconds. Once a message appears, the SP and LWP servers make it available at port 8091 for the next 10 seconds.

Clients of the SP server receive an "HTTP 204 No-Content" response in the absence of messages. Clients of the LWP server receive a connection reset in the absence of messages. In each case, the client receives an "HTTP 200 OK" response if a message is available, and the client sends a new request one second after either a reset or response.

S: At the second "S" phase, the server continues running for 120 seconds after the last message, with polling against SP and LWP, but no message delivery. The server is finally halted to allow the system to return to quiescent state before the next test execution is started.

These executions were done first using SP and LWP against a 3K population, and then using SP and LWP against a 30K population. This was the approach for all tests and associated graphs that follow, except where otherwise indicated.

LP Test

A single test correlating RAM with concurrent connections was done for the LP server with the 30K population. For the LP server, all clients connect immediately on startup and remain connected until the first message arrives. The LP server delivers the message immediately with "HTTP 200 OK" to these clients, which reconnect one second later to wait for the next message.

SP and LWP Graphical Output

Each graph of test results displays a particular metric for a given population size. Each graph includes both SP and LWP executions:

1. 3K clients, SP and then LWP
2. 30K clients, SP and then LWP

As annotated with Figure 2, the red markers in each graph indicate the startup times of SP and subsequently LWP servers, with messaging events indicated by the groupings of light-blue markers. While both populations are shown for the bandwidth metric in Figures 2 and 3, only the 30K population is shown for any metrics with similar results for both populations.

The selected set of metrics captured includes the following:

1. Network activity, to include bandwidth, socket use, TIME-WAIT and connection reset levels
2. CPU idle time
3. RAM and heap size usage

Sampling of the metrics was done at five-second intervals for finer-grained analysis. Graphs were displayed using minimum or maximum depending on the metric, with 20-second resolution for simpler visualization.

To facilitate understanding of overall test results and graphical presentations, reference the annotations in Figure 2. This can be consulted as a template for subsequent graphs in the TEST RESULTS section.

TEST RESULTS

Network Impact

Maximum kilobytes of input and output bandwidth are shown in Figure 2 for 3K clients and Figure 3 for 30K clients, with SP and LWP executions in each graph. Figure 2 provides detailed annotations of test phases.

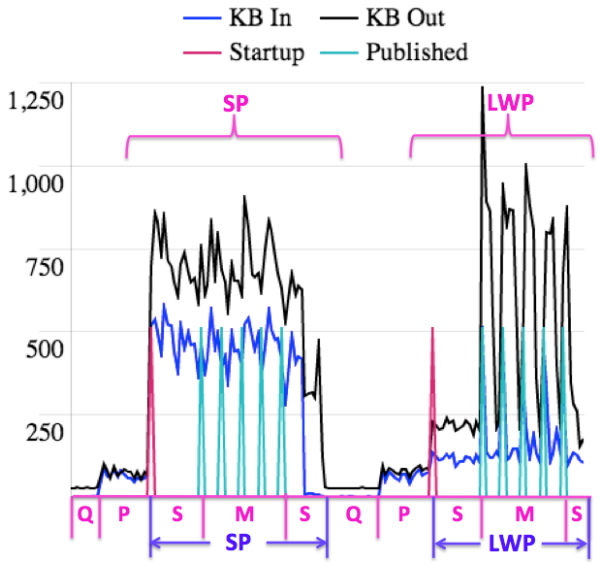


Figure 2: Bandwidth Impacts, 3K clients for SP and LWP executions

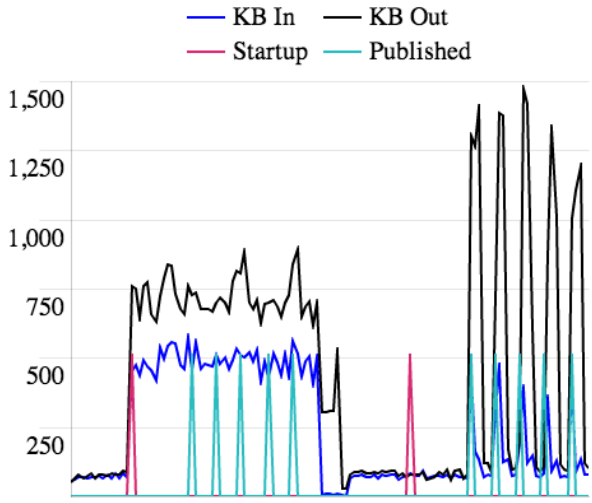


Figure 3: Bandwidth Impacts, 30K clients for SP and LWP executions

Variability in output peaks are likely due to the bumpy profile of client traffic. If capacity is to be provided for these infrequent peaks, it would be useful to reclaim capacity that is unused when there is no messaging. As illustrated in Figures 2 and 3, an SP environment exhibits bandwidth usage that varies in a narrow range near its highest levels, regardless of messaging traffic. This leaves less bandwidth margin to be reclaimed.

Alternately, bandwidth consumption for LWP stabilizes at about 40% of the SP levels in the absence of messaging traffic, engaging network capacity only when messages arrive. Additionally, the LWP bandwidth profile tapers shortly after the message event. This profile offers margin that can be applied to alternate capacity allocation. This margin increases as messaging frequency decreases. With an LWP system, this inverse correlation is independent of client population or polling rates.

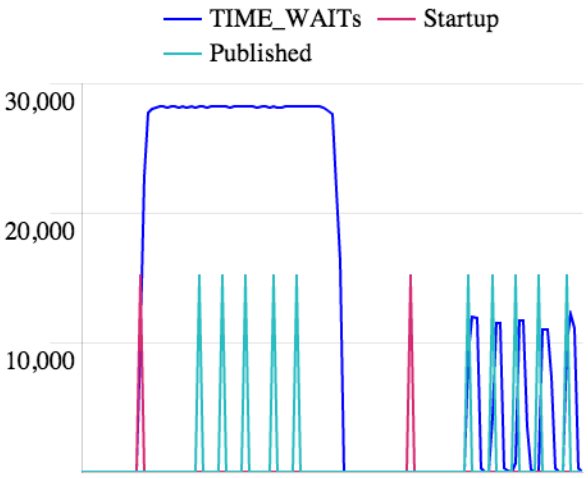
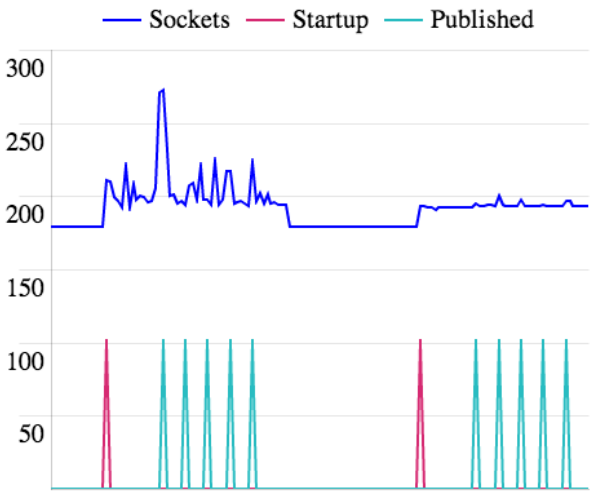


Figure 4: TIME-WAIT Activity, 30K clients for SP and LWP executions

The correlation with TIME-WAIT behavior can be seen in Figure 4. Because of configuration on the load generator machine, the limit on source IP-port tuples (which are stored in TIME-WAIT structures) was about 28K[13]. The chart shows that SP systems max out the potential TIME-WAIT levels for those connected source IP-ports immediately, keeping those levels high for its entire execution. However, LWP establishes fewer connections since the destination port is only

open for brief periods, exhibiting the same reduced peak and post-message tapering for TIME-WAIT as seen with bandwidth. TIME-WAIT activity can have impact on scalability and performance in various ways, as discussed in the FOOTNOTES section[10].



Figures 5: Total Sockets Profile, 3K clients for SP and LWP executions

Figure 6: Total Sockets Profile, 30K clients for SP and LWP executions

Maximum levels of sockets used by SP and LWP are shown in Figure 5 for 3K clients and Figure 6 for 30K clients. As with bandwidth, these peaks drive capacity planning. SP peaks are higher relative to LWP for both population sizes. LWP use of sockets remains slightly elevated above baseline independent of population size, peaking at higher levels for the 30K population. The difference in absolute levels of socket use with the 30K population is about 50% greater with SP vs. LWP; this suggests that LWP would scale more gradually (providing spare capacity in terms of sockets) as the population increases. More full-featured load generation would be needed for higher confidence.

The engagement of TCP connection reset activity using an LWP system accounts for the reductions in bandwidth, TIME-WAIT and socket levels. This is illustrated with Figure 7.

Figure 7: Connection Reset Activity, 30K clients for SP and LWP executions

A TCP connection reset can be triggered by various conditions[11]; in this case it is due to an unopened destination port. This happens immediately at test startup, as seen in Figure 7 preceding both red startup markers; this is because tests are begun with the polling-only baselines without the server running. Immediately on SP startup, the reset rate drops effectively to zero since SP has opened port 8091 and is returning “HTTP 204 No-Content” with no messages available. However, the resets continue after LWP startup since LWP waits until a message is available to open the destination port.

CPU Impact

The timeline in Figure 8 illustrates the minimum levels provided by SP and LWP for CPU idle percentage.

Figure 8: CPU Impact, 30K clients for SP and LWP executions

Available CPU declines immediately at SP startup time, with no benefit to the delivery of messages. Idle CPU remains below the 40% mark for the duration of the SP run for both populations. The CPU exhibits no noticeable reaction at LWP startup, remains high in the absence of messaging and has minimums higher than the SP maximums with messaging events. LWP engages the CPU only when delivery of a message is needed, and unlike an SP system, that CPU usage is transient with a quick recovery after the message.

It would seem LWP should drop to the same levels of idle CPU during message events as seen for SP, but it remains significantly higher. This is due to thousands of No-Content responses from SP to clients continuing to poll at one-second intervals. While SP engages the CPU to tell the client that there is no message, LWP systems avoid this wasteful activity.

The differences in CPU availability are shown with a frequency distribution of idle percentages, for SP and LWP in Figures 9 and 10 respectively. LWP never uses more than 50% of CPU, while SP sometimes consumes 100%.

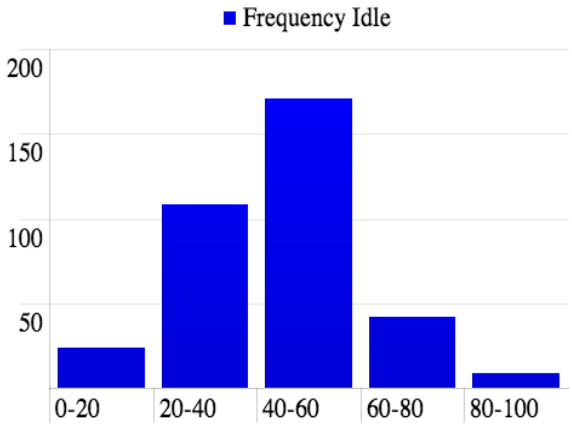


Figure 9: CPU Idle, Frequency Distributions, 30K clients for SP

With LWP there are many fewer 204 responses during messaging periods, since LWP closes its port at TTL expiration. This raises the question of why *any* 204 responses are seen, instead of all requests receiving either 200 OK or a connection reset. The reason is that the LWP implementation does not apply locks to the processing, allowing some requests into the request-handling pipeline while the port is open. By the time the last handful of these requests are processed by the handling logic, the TTL has expired and the message has been removed. In this case, as with the SP system, the response handler sends out the 204 responses.

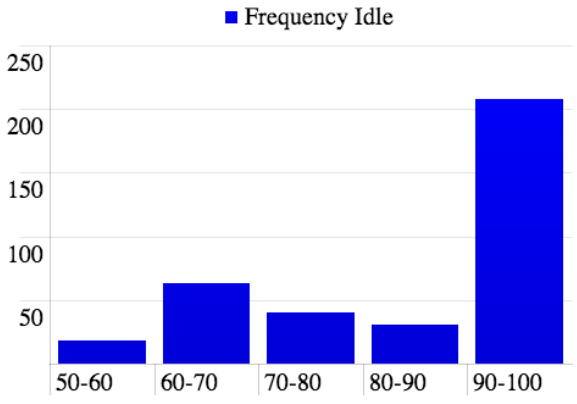


Figure 10: CPU Idle, Frequency Distributions, 30K clients for LWP

RAM Impact

There is only a slight difference in maximum RAM usage between SP and LWP systems, regardless of client size. Both plateau after all 30K clients have been engaged, as per Figure 11. The absolute values of RAM used by the SP and LWP systems are about 140MB and 170MB respectively.

Figure 11: Process Resident Memory, 30K clients for SP and LWP executions

Figure 12 shows the RAM levels achieved by the LP implementation for the same 30K population, displayed with the incremental increase in connections as clients are started in 3K sets. Connection levels here are multiplied by eight to more readily visualize the correlation. The peak RAM level approaches 30% higher than LWP (220MB). Note that the LP execution did not include any messaging events; it was done solely to determine the influence of additional connections on growth in RAM.

Figure 13 shows the incremental changes in RAM size as connections increase. The RAM cost for each connection averages about 9K. For all 30K connections the increase totals about 275MB, which is primarily unused capacity in an infrequent-messaging context. Note the cost of RAM per connection found here should not be regarded as typical, since implementations, system tuning and other factors will vary[1].

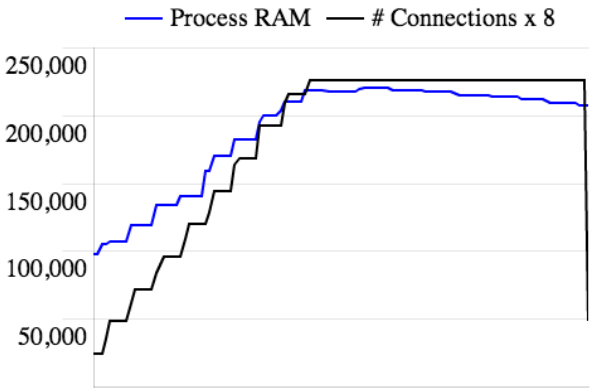


Figure 12: RAM Growth with Increases in # of Connections, 30K clients for LP only

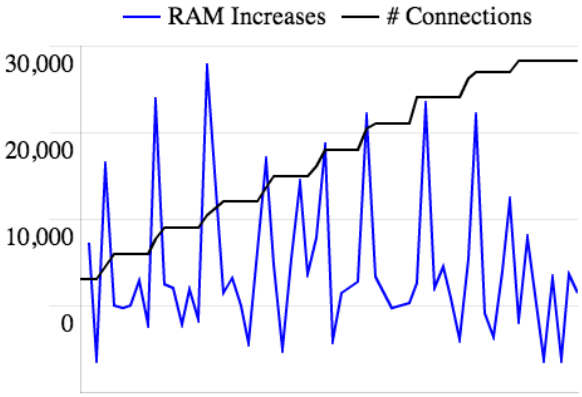


Figure 13: RAM Delta with Increases in # of Connections, 30K clients for LP only

Heap Size Impact

Java heap activity is shown using SP in Figure 14 and LWP in Figure 15, against the 30K population only.

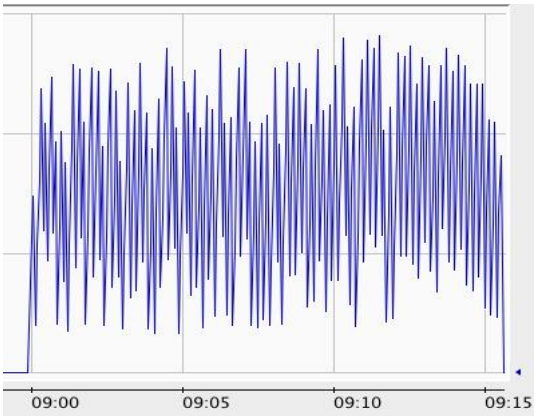


Figure 14: JVM Heap, 30K clients for SP

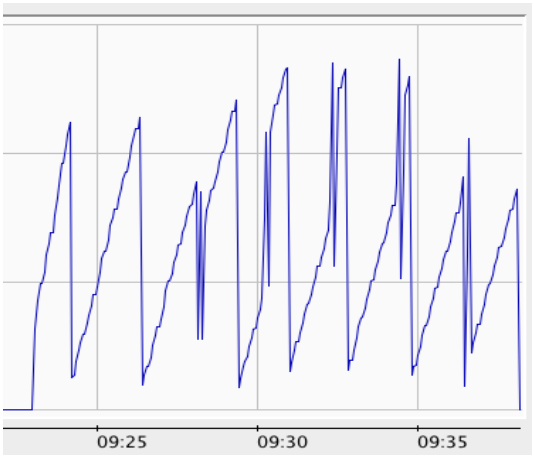


Figure 15: JVM Heap, 30K clients for LWP

Both heaps achieved approximately the same size, but this is a function of JVM and garbage collection tuning parameters. The variations in heap size appear to be a result of garbage collection activity, which is known to have measurable impacts on system response[12]. There is considerably less of that activity in the LWP system.

Another test execution over a longer period compares heap activity for SP in the first half, followed by LWP in Figure 16. SP has a great deal more activity but is losing ground over time, while LWP shows more desirable behavior. Additional testing would be useful here to capture not only heap size, but also generational activity and pause times for a more insightful analysis.

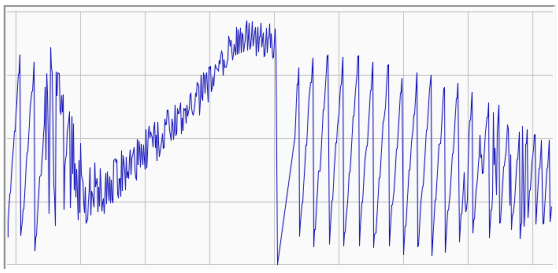


Figure 16: JVM Heap Size, 30K clients for SP and LWP executions

CONSIDERATIONS

All tests were done with HTTP (no SSL).

LWP Tradeoffs

LWP has many considerations, to include at least the following:

1. A specialized server is required.
2. This is prototype technology, barely emerging from proof-of-concept stages.
3. Clients must be aware that they will receive frequent connection resets.
4. Clients cannot determine if the service is down since connection resets are the norm. That said, clients – including load balancing systems that manage failover – can execute health checks against LWP at port 8088, which is open permanently. See the Class Diagram in the APPENDIX for details.
5. If SSL is used and the load balancer handles SSL offload, then there is an undesirable impact against that network component. This undermines the original purpose of LWP.
6. If SSL is used but the load balancer does not handle SSL offload, then load balancer

caching cannot be leveraged. This would call for SSL offload at LWP and a shared cache instead, assuming horizontal scale for LWP servers.

7. If SSL is used but the load balancer does not handle SSL offload, then routing decisions, session management based on headers, and any other application-layer functionality provided by the load balancer cannot be leveraged.
8. Messages must be cached for a TTL of at least the expected latency tolerance of the clients. However, LWP is intended only for "strict latency" environments, so that TTL is expected to be short.
9. If the message TTL is longer than the polling interval, the *port-per-message* protocol (which is to-date undeveloped) should be used. Otherwise, clients polling at high frequency will unnecessarily burden LWP, and these clients must distinguish duplicates.

The points in items 5-7 raise a question of whether LWP-type logic should live in a proxy server acting as SSL offloader, load balancer and centralized cache.

CONCLUSIONS

The architectural goal of LWP is to use resources only when needed to deliver messages, releasing that capacity for other purposes in their absence. Tests results show that LWP makes low-impact demands on network and CPU, moderating various undesirable characteristics of traditional short poll systems, and that it is well suited for strict latency, infrequent messaging contexts.

LWP also addresses the unused commitment problem (dedicated but idle capacity) of long poll systems in the strict-infrequent context. This justifies a second look at LWP-based short polling as an alternative to long poll for use cases calling for that context.

In short, LWP accomplishes this by opening a TCP port only when the message is available. With this approach, data center provisioning is now primarily driven by messaging frequency rather than client size or latency requirements. The resources supporting an LWP system are increasingly available for other activity as messaging frequency decreases.

Conversely, with increase in messaging frequency, LWP performance degrades to more closely resemble traditional short poll. The increasingly leveraged capacity of long poll provides more favorable cost-benefit in this context.

As with many technology solutions, LWP brings trade-offs and considerations. The most suitable use is narrow: strict latency requirements with infrequent messaging. The prototyping done to-date has moved beyond proof-of-concept to demonstrate promising performance results, but LWP remains for now a research project with numerous open questions to be resolved.

APPENDIX

Class Diagram

For reference at the implementation level, an overview class diagram describing responsibilities and collaborations is given here.

The *ContentListener* is the component that opens a port permanently (8088 as tested), listening for both Client subscriptions and messages from the Publisher. The *ContentListener* is responsible for creating one *TopicHost* for each supported topic.

The *TopicHost* component opens a port (8091 as tested) for its assigned topic only when a message is available (and only for a brief period as specified in the TTL). *TopicHost* delegates to the *MessageSender* to handle incoming client requests for that message.

Since the *ContentListener* always listening, its port should be used for health checks instead of the *TopicHost* port, which is usually closed.

The *port-per-message* protocol has not to-date been implemented; only *port-per-topic* is shown here.

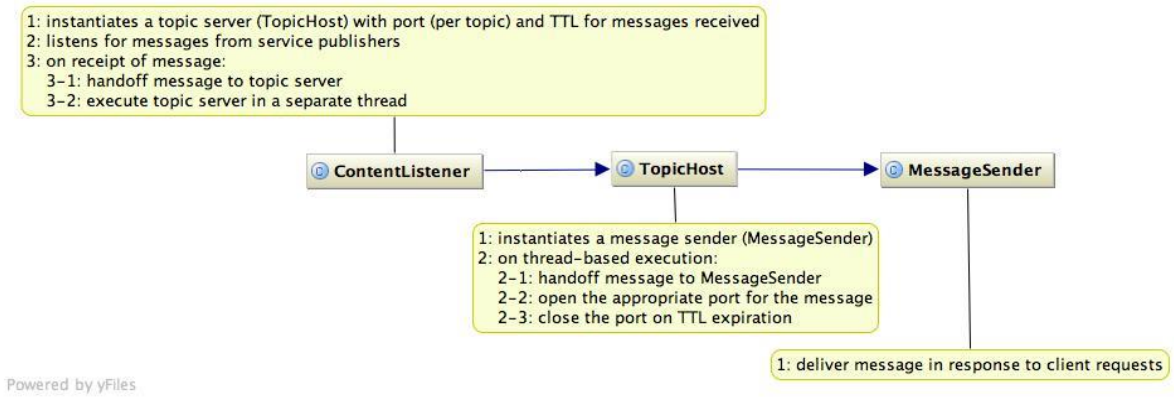


Figure 17: Class Diagram for netty Implementation of LWP

FOOTNOTES AND REFERENCES

[1] TCP Cost

http://en.wikipedia.org/wiki/Transmission_Control_Protocol#Resource_usage
http://stackoverflow.com/questions/7669293/performance-implication-of-creating-new-tcp-connection-per-message
http://stackoverflow.com/questions/4139379/http-keep-alive-in-the-modern-age
http://stackoverflow.com/questions/4840116/general-overhead-of-creating-a-tcp-connection
https://tools.ietf.org/html/rfc955
http://en.wikipedia.org/wiki/Transmission_Control_Protocol#Resource_usage
http://stackoverflow.com/questions/3173720/keeping-1000-tcp-connections-open-inspite-of-very-few10-20-actual-communicatio

TCP setup involves a three-way handshake, with teardown requiring a four-way exchange, adding latency to each connection. The size of each transfer is typically small enough to fit into one network packet. While network latency and impact on network stacks (CPU and kernel) comprise the majority of cost, bandwidth use is proportional to client population size. Estimates from others, as seen in some of the discussions referenced above, suggest varying levels of RAM cost per connection.

[2] COMET

http://en.wikipedia.org/wiki/Comet_(programming)

[3] Long-poll and Streaming Considerations

http://tools.ietf.org/html/draft-loreto-http-bidirectional-07#section-2.2

[4] WebSockets

http://en.wikipedia.org/wiki/WebSocket
http://www.websocket.org/
http://www.whatwg.org/specs/web-apps/current-work/multipage/network.html

[5] Options to Support "graceful-scalability"

http://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html#sec8.1
http://web.archive.org/web/20100813132504/
http://www.io.com/~maus/HttpKeepAlive.html
http://users.cis.fiu.edu/~downeyt/cgs4854/timeout
http://tools.ietf.org/id/draft-thomson-hybi-http-timeout-01.html
http://stackoverflow.com/questions/4139379/http-keep-alive-in-the-modern-age

Connection Keep-Alive is used by default in HTTP 1.1, but the customized clients and servers used here would require additional steps to configure it. Those steps were not done for the tests done here since the use case under test assumes infrequent messages. Given that assumption:

- Applying Keep-Alive to SP would blur the distinction with LP and its unused capacity problem;
- Applying Keep-Alive to LP would moderate reconnect cost, but was left undone since only the impact of persistent connections on RAM (vs. latency and CPU usage from setup and teardown) was measured for the LP implementation; and
- Keep-Alive is irrelevant for LWP by virtue of that system's overall strategy (no connections are made without messages)

HTTP Pipelining is an interesting option that can be revisited in follow-ups, but was omitted here to reduce the number of test permutations.

[6] TCP Reset Sequence

http://blog.creativeitp.com/posts-and-articles/networking/exploring-idle-scanzombie-scan/
http://en.wikipedia.org/wiki/Transmission_Control_Protocol

When a port is not open, as done with LWP by design, the attempted setup consists of a SYN from the source peer, followed immediately by RST from the destination peer. This terminates connection setup immediately.

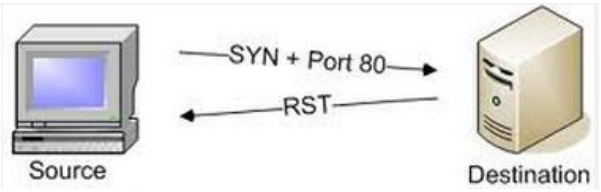


Figure 18: TCP Connection RST Sequence

[7] TCP Setup/Teardown Sequences

http://blog.creativeitp.com/posts-and-articles/networking/exploring-idle-scanzombie-scan/
http://en.wikipedia.org/wiki/Transmission_Control_Protocol

TCP setup resulting in a connection consists of three exchanges: the client sends a SYN to start the dialog, the server responds with a SYN, ACK, and finally the client sends an ACK. From that point the connection is in the ESTABLISHED state. Teardown uses four exchanges, with FIN from terminating peer, ACK and FIN from the remote peer, followed by an ACK from the terminating peer.

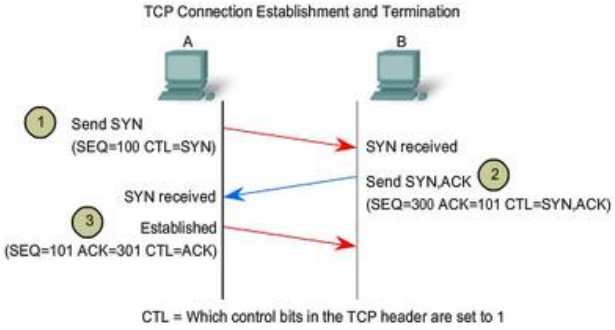


Figure 19: TCP Connection Setup Sequence

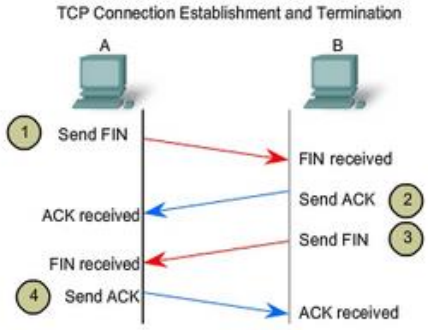


Figure 20: TCP Teardown Sequence

[8] netty

http://netty.io/
http://en.wikipedia.org/wiki/Netty_(software)

[9] Port Capacity on Linux

http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html
http://www.nateware.com/linux-network-tuning-for-2013.html

Linux systems are configured to allow ports with numbers ranging from 32768 to 61000 by default, whether ephemeral (outbound from clients) or assigned to applications (inbound to servers), as managed using `sysctl` with `net.ipv4.ip_local_port_range`. This range can be increased up to 65535, and possibly as low as 1024 depending on ports reserved for system services.

[10] TIME-WAIT

http://www.ncftp.com/ncftpd/doc/misc/ephemeral_ports.html
http://www.lognormal.com/blog/2012/09/27/linux-tcpip-tuning/
http://www.nateware.com/linux-network-tuning-for-2013.html
http://www.hjp.at/doc/rfc/rfc3102.html#sec_6.1
http://www.isi.edu/touch/pubs/infocomm99/infocomm99-web/
http://www.serverframework.com/asynchronousevents/2011/01/time-wait-and-its-design-implications-for-protocols-and-scalable-servers.html

The duration of TIME-WAIT (typically between 60-240 seconds by default) prevents the same client (source IP) from connecting to the same service (destination IP-port) using the same ephemeral port (source port). IP stacks will typically allocate different ephemeral ports for the next connection request from that client, but as the ephemeral port range decreases and the client polling frequency increases, it becomes more possible to run out of ephemeral ports and to be unable to establish a connection.

TIME-WAIT on Linux systems can be managed using `sysctl` with the parameter `net.ipv4.tcp_fin_timeout`. On the system under test in experiments done here, the TIME-WAIT timeout was 60 seconds.

[11] Causes of TCP Resets

http://stackoverflow.com/questions/251243/what-causes-a-tcp-ip-reset-rst-flag-to-be-sent
http://myaccount.flukenetworks.com/fnet/en-us/supportAndDownloads/KB/IT+Networking/protocol+expert/What_are_TCP_RST_Packets_-_Protocol_Expert
http://blogs.technet.com/b/networking/archive/2009/08/12/where-do-resets-come-from-no-the-stork-does-not-bring-them.aspx

[12] Java Heap and Garbage Collection

http://javabook.compuware.com/content/memory/analyzing-java-memory.aspx

[13] Socket and Connection Capacity on Linux

http://www.cyberciti.biz/faq/linux-increase-the-maximum-number-of-open-files
http://www.lognormal.com/blog/2012/09/27/linux-tcpip-tuning/
http://www.nateware.com/linux-network-tuning-for-2013.html

On Linux systems, maximum sockets and connections can be configured. The relevant parameters for connections are under `net.netfilter` and include the current count `nf_conntrack_count` and the maximum `nf_conntrack_max`, as managed by `sysctl`. The maximum number of connections configured for the system under test in these experiments is about 64K.

As these limits increase, kernel memory usage also increases. Sockets are also limited by number of maximum open file handles, as managed with `fs.file-max`.

While the server could have accepted 64K connections, the number of (outbound) sockets for the load generator in these experiments was limited by its ephemeral port range of 28K (as per the default range of 32768 to 61000). This can be revisited in subsequent testing to generate more client threads, but more useful effort would go towards establishing multiple source IP addresses instead.