Performance Evaluation of an Alert Dissemination Engine based on the AT&T Enterprise Messaging Network

Huale Huang, Yih-Farn Chen, Matti Hiltunen, Rittwik Jana,
Serban Jora, Radhakrishnan Muthumanickam, Ashish Singh
AT&T Labs - Research
180 Park Ave, Florham Park, NJ, 07932, USA
email: {huale, chen, hiltunen, rjana, jora, rmuthu, ashks}@research.att.com

Abstract

The recent surge in the variety and number of mobile devices used as communication end points has created a significant challenge for messaging applications that aim to reach their target recipients regardless of their location and available devices. Alerting services used to notify a potentially large number of recipients about an emergency, or other important events, are an important class of applications enabled by the prevalence of such mobile devices. A middleware platform that arbitrates content delivery and adaptation between mobile devices and backend messaging applications is crucial in reducing the software complexity on both the client device and server side applications. For an alerting application, the Quality of Service (QoS) of the middleware platform becomes crucial - alerts must be delivered quickly, reliably, and securely to all their recipients. As the first step in achieving such QoS, this paper evaluates the performance of a commercial mobile middleware platform, the AT&T Enterprise Messaging Network (EMN), in the context of an alerting service. EMN is used as an Alert Dissemination Engine (ADE) to provision users, disseminate alerts, collect acknowledgments, and prepare reports on the status of alert dissemination and acknowledgments. The evaluation is based on a combination of component benchmarking and end-to-end benchmarking.

1 Introduction

Alerting emergency personnel, as well as ordinary citizens, of events such as accidents, terrorist attacks, or natural events (e.g., tsunamis or tornadoes) is quickly becoming possible thanks to the prevalence of various types of mobile devices. However, constructing an alerting service that delivers alerts to these various devices in various forms (voice, text, etc.) using various protocols (e.g., SMS, email, fax, pager) quickly and reliably is a significant challenge. This paper introduces such an alerting service (Alert Dissemination Engine - ADE) based on the AT&T Enterprise Messaging Network (EMN) [1][1] service and provides preliminary results on the performance aspects of the platform.

Performance is critical for an alerting service such as ADE especially when dealing with public safety. The number of people to be alerted may potentially be in tens of thousands or even more, multiple alerts may be sent concurrently using the same service, and other functions such as alert acknowledgment collection and reporting also consume the computing resources of the service. Therefore, the service implementation must be efficient, scalable, and predictable from the standpoint of performance of a given hardware configuration of the service. However, EMN is a relatively complicated system consisting of COTS components such as databases, message queue providers, firewalls, web servers, and mail servers, as well as proprietary components developed in-house. The job of processing and disseminating an alert touches most of these components.

It is important to note that ADE depends on external providers to complete end to end alert dissemination. For example, delivering a voice call requires the existence of the public switched telephone network (PSTN) to complete the outbound leg of the call. Results obtained in this paper (dissemination durations etc.) via a series of benchmarking tests have taken into account these constraints and where possible simulated these external constraints (exponential inter-arrival times etc.). Specifically for email delivery, a sink was constructed within the same high speed test network so as to justify timing only the delays experienced within ADE and not due to the variable propagation delays experienced between ADE and a real email sink across the Internet. This can be interpreted as a lower bound for dissemination times.

The performance objective of the ADE design is to

[1]EMN was formally known as *iMobile-EE* or *AT&T Mobile Network*.

"push" messages through the service to the alert recipients as quickly as possible with the lowest possible overhead. Towards this objective we have considered several optimizations within ADE for high throughput, low end to end latency, and resilient delivery mechanisms. In this paper, we discuss some of these optimizations, including *alert splitting*. In alert splitting, the ADE client segments a very large alert into smaller segments prior to submission to ADE. The results indicate that splitting a very large alert into smaller segments can be handled much more efficiently since it benefits directly from the load balancing between the servers.

This paper is organized as follows. Section 2 provides a quick system overview. Section 3 presents the performance testing strategies followed by numerical results. Section 4 highlights a specific optimization technique, alert splitting, used to reduce the total alert dissemination time. Section 5 discusses future optimizations and related work.

2 System Overview

2.1 AT&T Enterprise Messaging Network

AT&T Enterprise Messaging Network (EMN) consists of a number of proprietary components, namely EMN *gateways* and *servers*, and a number of off the shelf software components such as databases, mail servers, web servers, and JMS (Java Message Service)[4] servers. All of these components are replicated for load balancing and fault tolerance. EMN gateways are responsible for providing device and protocol-specific interfaces for different mobile devices to access services provided by EMN. In addition, gateways perform authentication, device profiling, and session management functions. EMN provides gateways for a multitude of protocols: email, http, pager, voice, fax, SMS, telnet, and instant messaging. EMN servers are responsible for implementing the services provided by EMN by performing local processing and accessing information sources. Each EMN service is implemented by a component called *infolet* that executes in the EMN server. Infolets implement the associated application logic and usually provide access to one or more sources of information. We distinguish the notion of services and infolets. Infolets are targeted towards fulfilling user related interactions (e.g.,. automatic destination device transcoding support) whereas *services* offer a programmatic interface to a set of functionalities. To be specific ADE comprises of a set of services mainly provisioning, alerting, reporting and injection [3].

The overall EMN architecture is presented in figure 1. Service requests to EMN arrive at the various gateways, the gateways use database to authenticate the requests and then forward them to JMS servers. The EMN servers pick up requests from the JMS servers and process them locally or use external information sources, and send replies back to the JMS servers. The gateways pick up replies from the JMS, match them with requests, and forward the replies to the correct user. Note that the whole system is protected by firewalls and load balancers are used to balance the incoming load between different http and mail gateways. Also, the mail and pager gates use external mail servers for incoming and outgoing requests and replies and voice gateways use VoiceGenie[6] servers. The fax and SMS gateways use external service providers to deliver outgoing messages.

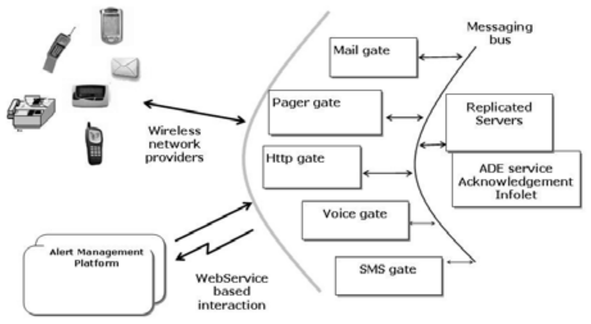
Messaging bus
Mail gate
Pager gate
Http gate
Voice gate
SMS gate
Replicated Servers
ADE service Acknowledgement Infolet
Wireless network providers
Alert Management Platform
WebService based interaction

Figure 1. EMN Architecture

2.2 Alert Dissemination Engine

The Alert Dissemination Engine (ADE) is a service built on EMN as a collection of infolets. Specifically, ADE allows an authorized person to submit an alert that will be delivered to the different devices of the registered users. The service allows the users to acknowledge the reception of the alert and allows the person submitting the alert to keep track of which of the intended targets have received and acknowledged the alert.

ADE consists of three infolets: provisioning, notification, and acknowledgment. The provisioning infolet allows an authorized user to add, delete, and edit alert recipients and their devices. The notification infolet allows an alert to be submitted. The alert is specified as an XML-document that contains the identifiers of the intended recipients, the alert message to be delivered, and delivery constraints that provide the system additional details on how the alert should be processed. The recipients can be specified on different levels of granularity ranging from a predefined group identifier to a specific user's specific device address (e.g., cell phone number). The alert content may be provided in different formats for different devices, for example, SMS messages are limited to 160 characters. The issuer of the alert may also register to be reported about the progress of the alert dissemination. This is done by providing a call-back address as a part of the alert submission. ADE will then send periodic reports as XML-documents. Finally, the acknowledgment infolet allows a user to acknowledge the reception of an alert.

Alert dissemination is the most performance critical function provided by ADE. Therefore, in the rest of the paper, we will focus on alert dissemination as well as reporting and acknowledgment since they usually overlap with the dissemination for large alerts. However, since provisioning typically happens before the system is used for real alerting, we will not consider provisioning in any more detail. The processing of an alert is split into two phases: solving and performing. Since alert processing may take considerable amount of time, EMN provides two modes for submitting an alert. In the synchronous mode, the alert is received by EMN and solved before a reply is sent to the alert issuer. In asynchronous mode, EMN simply stores the alert request in a database and returns an alert identifier to the alert issuer. This identifier can be used to request the status of the alert and register for reports.

Alert solving includes the verification and expansion of each of the alert recipients into protocol-specific end point addresses, for example, the phone number or the email address of a person is an end point. The verification of each recipient involves a database lookup. The recently-checked recipients are cached by the EMN server to reduce database access in case many alerts are sent to the same receivers. Furthermore, alert solving involves the analysis of the delivery constraints of the alert and construction of a report structure for the alert. Alert dissemination is executed by a series of performers. For every alert there are four kinds of alert performers: trigger, acknowledgment, timer, and reporter. All performers define a set of event types they trigger inside the engine: for example, trigger events notify about the execution of the notifications. Trigger performer takes the input of the trigger solver, a set of trigger procedures, and runs them through a planner which decides on the timing of the dissemination process. During the notification, after sending the forward request towards the gateways, it updates the alert report entries with the current status. ACK performer monitors every acknowledgment that is requested and updates the acknowledgment report entries. Reporter is responsible for the management of the alert observers and the notifications they registered for. It has its own built-in solver that handles the reporting scheme by subscribing to the required events produced by the other performers. The reporter manages the report persistence procedures during the alert lifetime.

Each outgoing alert message flows from the EMN server through a JMS provider to an appropriate gateway. Each gateway uses its specific technique to deliver the alert message to the user. For example, mail gateways construct an email message out of the alert and send it to a local COTS mail server. This mail server will then use the standard SMTP protocol to deliver the email message to the recipient's mail server. The voice gateway uses VoiceGenie to initiate an outgoing phone call to the recipient's phone number. SMS gateway uses the standard SMPP (Short Message Peer to Peer) protocol to interface with an external SMS broker for sending and receiving SMS messages. Pager gateway uses standard SMTP or SNPP (Simple Network Paging Protocol)[2] protocols to send and receive pager messages through paging carriers. The selection of SMTP or SNPP protocol is based on a device's addressing scheme and paging carrier's interface support.

3 Performance Testing

Our EMN performance testing is based on the combination of *component benchmarking* to determine the performance characteristics of individual components and *end-to-end benchmarking* to determine the overall system capacity under different application scenarios.

3.1 Component Benchmarking

Benchmarking a component requires our test environment to be set up in such a way that there are no bottlenecks in other components that would affect the measurement of the target component. The EMN architecture allows individual components to be replicated for increased performance and reliability. Component benchmarking helps us determine when to add additional instances of a component to satisfy our Quality of Service (QoS) requirements. The individual components include the EMN gateways (http, email, voice, fax, pager, and SMS), EMN server, Voice Genie, SMTP server, JMS Server, and the Oracle 9i database server.

Our component benchmarking testbed consists of one SunFire V240 running SunOS 5.8, one VoiceGenie server, and four Dell 1750s, two Dell 650s, and two Dell 350s, all running Red Hat 9.0 Linux. Two of the Dell 1750s are used as the gateway machines with each of the six EMN gateways (one for each protocol) installed. One Dell 1750 was used to run one instance of the EMN server. The Oracle database and Sonic JMS run on the SunFire machine. The two Dell 650s run two SMTP servers and the two Dell 350s run a fault-tolerant load balancer (LVS [7]).

3.1.1 Benchmarking the HTTP Gateway

Our target matrix for the HTTP gateway is

- *(SOAP alert msg size, number of concurrent alerts)* $\rightarrow$ *HTTP gateway throughput (requests/sec)*

The message size of a typical HTTP SOAP alert request that posts an alert depends on the number of endpoints in

Number of endpoints	*Message size in bytes*
32	49,362
128	191,281
512	764,920
2048	3,071,499

Table 1. Number of endpoints vs. message size

each alert (see Table 1). As we can see from the table, the overhead caused by the SOAP message structure can be significant due to the linear dependency between the number of endpoints and SOAP message size: a typical alert with 2K endpoints can result in a 3MB SOAP packet. Note that the JMS message size limit is 1MB (as recommended by the vendor). To reduce the communication overhead, the HTTP gateway compresses each SOAP request before sending it through JMS and the EMN server decompress the request before processing it. The typical compression ratio is roughly 20 to 1.

Listing all endpoints explicitly is only one instance of the various mechanisms that can be used to specify an alert. Alternatively, we can use the group specification (i.e., group-id:/channel) to specify an alert. In this case, the overall SOAP message size stays constant regardless of the number of endpoints associated with a group. This helps to specify an alert in a more compact manner.

To measure the raw HTTP gateway performance, we use a dummy EMN server that simply echoes *success* when an alert request arrives. We use a test client that simulates various numbers of threads and endpoints, with each thread sending 100 requests with the Poisson distribution (mean=150ms) used for the request arrival interval. Intuitively, a thread simulates an *affiliation manager (AM)*, who is allowed to post alerts. For a typical emergency scenario, we assume that an AM will send at most 100 successive requests. During alert splitting (see Section 4), a large alert sequence may also be simulated by multiple sequential smaller alerts.

We measured the following parameters in the experiment:

- average request response time (ms)
- total execution time (ms)
- throughput (requests/sec)

Partial results are shown in Table 2, which shows that the HTTP gateway achieves maximum throughput (57 requests/sec) for alerts of 16 endpoints when there are 40 concurrent threads given the request distribution. Note again that the focus of this measurement is on the raw HTTP gateway throughput, as a dummy server was used and no email dissemination time was measured. End-to-end results are available in Section 3.2.

Number of alert requests	*Number of endpoints*	*Mail dissem. time (s)*	*Throughput (mails/s)*
50	200	612	16.34
50	400	939	21.30
100	200	971	20.60
100	400	1807	22.14

Table 3. Throughput of Mail Dissemination

3.1.2 Benchmarking the Mail Gateway

Our target matrix for the Mail gateway is

- *(number of endpoints, message size)* $\rightarrow$ *mail throughput (mails/sec)*

To measure the throughput of mail push, we use only one mail gateway connected to one SMTP server. We also set up 400 user accounts on a separate SMTP server acting as receiving clients. No acknowledgments are sent back for the initial experiment.

Another test client is set up to fire alerts (sequences of 50 or 100) with 200, or 400 mail endpoints continuously. Size of each email alert is 702 bytes. Table 3 shows the test results.

After examining the mail gateway logs and the SMTP server logs, the bottleneck appears to be at the SMTP server (a Dell 650 server) with a maximum throughput of 22 messages/sec. Further investigation is necessary to see whether it's the hardware or the SMTP server configuration that limits the throughput.

3.2 End-to-End Email testing

For end-to-end benchmarking, we focus on the most typical application scenario, which consists of sending a typical alert to different numbers of end points with and without the alert acknowledgment requirement.

For email alerts, we use our existing staging environment for performance measurements of alert posting, mail dissemination, and acknowledgment processing using the following variable *input* parameters:

- Email alert message size: 2KB (typical)
- Number of client threads: 1, 2, 4, 8, or 16,
- Number of end points in each alert: 32, 64, 128, 256, 512, 1024, 2048, or 4096.

Number of endpoints	*Number of threads*	*Number of requests*	*Average response time (ms)*	*Total time (ms)*	*Throughput (requests $\times$ threads/s)*
16	1	100	48.57	22815	4.383
16	10	100	71.70	24311	41.134
16	40	100	523.95	70163	57.010
16	160	100	10163.74	1292164	12.382
64	1	100	334.50	50146	1.994
64	40	100	10246.26	1045938	3.824
64	160	100	31233.84	4079899	3.922
128	1	100	610.67	77807	1.285
128	40	100	19612.86	1986795	2.013
128	160	100	61218.81	8082589	1.980

Table 2. Measurements of HTTP gateway throughput

Scenario set	*Request interval*	*Number of threads*	*Number of endpoints*
S1	1 min	1,2,4,8,16	32,64,128,256,512
S2	5 min	1,2,4,8,16	1024,2048
S3	10 min	1,2,4	4096,8192

Table 4. Scenario Sets for End-to-End Mail Dissemination Test

- Mean request arrival interval: 4, 8, 16, or 32 minutes used in the Poisson distribution
- Number of successive requests: 1 or 5.

Note, however, that it's not realistic to have a very short mean request arrival interval when the number of endpoints is in the order of 4K - given the kind of QoS guarantees that we are willing to provide for our platform. Table 4 shows the *scenario sets* that we consider to be *realistic*.

Here are the output parameters that we would like to measure:

- Min, Mean, Max and variance of initial HTTP(s) resonse times for each scenario and ensemble.
- Total mail dissemination time: compute throughput for each scenario and ensemble
- Acknowledgment processing: Measure both cases when the receiving client acknowledges each arriving mail alert immediately or with a random delay - with a mean of 5 minutes before responding.

Table 5 shows some partial results of our initial measurements. For all these tests, only one request per thread was issued. The results show a clear improvement in throughput as the concurrency (number of threads) is increased. It also shows the tradeoff between average response time and throughput. As the latter increases, the delay in the HTTP gateway response time (to confirm the receipt of an alert posting request) for each thread is increased. Note that the test environment has two HTTP gateways, two mail gateways, and two SMTP servers, but only one EMN server and only one primary Oracle database. The end-to-end throughput can be limited by either the single EMN server, the database, or the slow SMTP servers (with a total capacity of roughly 40-44 mails/s).

4 Alert Splitting

Alert splitting means that the EMN client breaks up alerts with large number of end points into multiple smaller alerts, segments, with smaller number of end points. Alert splitting not only alleviates issues with resource capacity, but also improve the overall performance of the EMN platform. Very large alerts cause a number of problems, including alert messages too large for JMS (JMS package size currently set to 1MB), too long processing time (HTTP timeout on the EMN client side is currently set at 90 seconds), and extensive memory usage at the EMN gateways and servers. Splitting an alert not only alleviates these issues, but it can also improve the alert processing and dissemination time because the alert is processed concurrently at multiple EMN servers.

The responsibility for alert splitting lies at the EMN client (such as CHAIN, an Alert Management Platform used in the CHAIN-EMN service[5] provided by AT&T). The EMN client submits each alert segment as a separate alert for EMN and maintain their correlation. EMN handles each alert segment as an individual alert and does not know that these segments belong to one large alert. Specifically, EMN creates a different ID for each such segment whereas the EMN client uses one alert ID for all the segments in one alert. EMN provides separate reports for each segment and the EMN client is responsible for merging these reports based on their alert IDs. The EMN client should also

Number of endpoints	*Number of threads*	*Avg. response time (ms/thread)*	*Total response time (ms)*	*Mail dissemination time (ms)*	*Throughput (mails/s)*
32	1	1416	1416	5907	5.42
32	4	2009	2605	13015	9.83
32	16	7703	8393	19903	25.72
128	1	3024	3024	16868	7.59
128	4	5315	6996	27933	18.33
128	16	20764	22779	68019	30.11
512	1	12710	12710	38658	13.24
512	4	16134	22096	119125	17.19
512	16	88155	92702	292247	28.03
1024	1	37225	37225	396299	2.58
1024	4	57689	70747	277241	14.77
1024	16	293606	294482	681984	24.02

Table 5. End-to-End Mail Dissemination Measurements with a single EMN server

build the application logic for handling partial errors, e.g. if one of the segments fails during submission, other segments may continue to function on the EMN side.

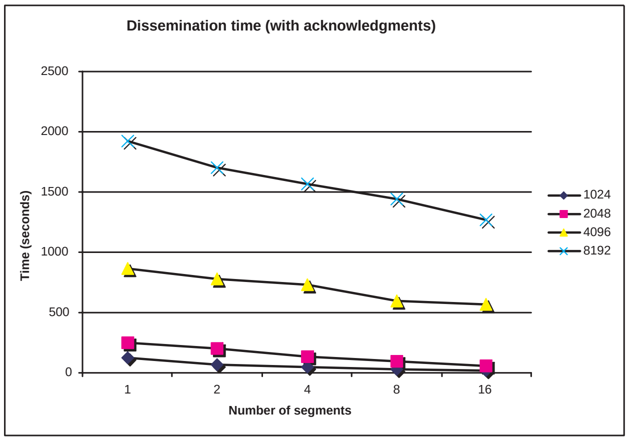

Figure 2. Alert dissemination time with acknowledgments

We performed a number of experiments to illustrate the effectiveness of alert splitting and to attempt to determine the optimal segment size. The alert processing and dissemination time is expected to reduce when the alert is split into few segments, but should eventually start growing if the alert is split into too many segments since EMN must maintain information for each segment separately and must generate reports to each segment. For our experiments, we used an alert message of 2 KB and disseminated this alert to 1024, 2048, 4096, and 8192 email endpoints using different segment sizes. The reporting frequency was set to 30 seconds and the client submitted all the segments in an alert sequentially using asynchronous alert submission with no extra delay between segments. We performed the experiments with and without acknowledgment constraint.

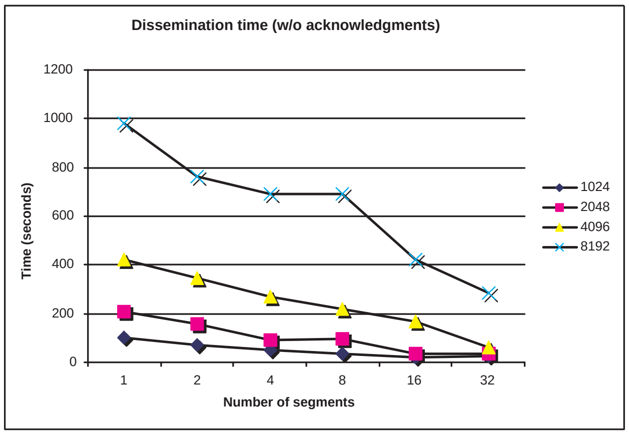

Figure 3. Alert dissemination time without acknowledgments

The impact of alert splitting on the total alert dissemination time is presented in figures 2 and 3. The figures show the total alert dissemination time from the time the first segment is submitted to EMN until the last alert message is delivered to the last receivers mail box. In general, the figures show that alert splitting has a significant impact on the alert dissemination time, but the benefit of splitting an alert into smaller segments reduces when the number of segments gets large and the individual segments get very small (e.g., 32 end-points in one segment).

Acknowledgments have a significant impact on alert dissemination time with and without alert splitting. In general, alert dissemination takes considerably longer when alert acknowledgments are required. For example, disseminating an alert with 8192 end points without splitting takes almost 2000 seconds when acknowledgments are required, while it takes less than 1000 seconds if they are not required. This is because the acknowledgment messages are processed by the same EMN gateways and servers that are involved in disseminating the alert notifications and for large alerts, the acknowledgment processing occurs concurrently with the alert dissemination. Furthermore, we note that while alert splitting reduces the dissemination time with and without acknowledgments, the impact is greater for alerts without acknowledgments.

We also calculated the throughput of the system in terms of alert end-points delivered in a time unit. This metric is calculated simply as the number of end points in an alert divided by the total alert dissemination time. This throughput is plotted in figures 4 and 5. These throughput figures illustrate that alert splitting has a large initial impact on small alerts, but the system gets eventually saturated and further splitting can reduce throughput (see figure 5). For larger alerts with acknowledgments, splitting has a smaller impact on the overall throughput since the system resources are close to saturated even without splitting.

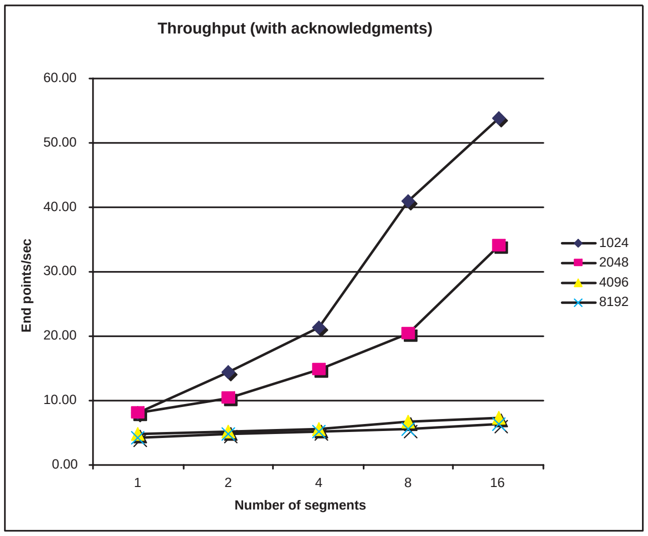

Figure 4. Alert processing throughput with acknowledgments

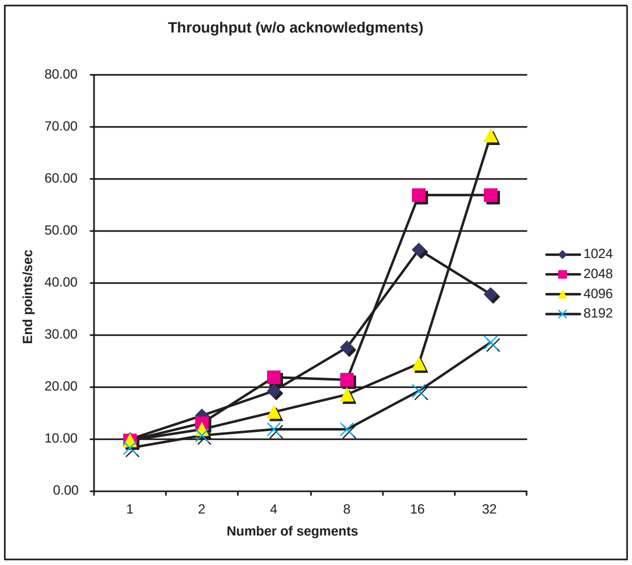

Figure 5. Alert processing throughput with no acknowledgments

5 Discussion and Insights

Dissemination performance depends on various factors namely the detail to which an alert is specified, the QoS requirements and the required levels of personalization. The way the alert is specified affects directly the request size, the endpoint solving/expansion process (number of DB requests) and the alert solving process (one global level constraint applying to all endpoints vs. similar constraints repeated for each endpoint). This influences primarily the response time for an alert submission and increases the overall load of a server (i.e., more processing cycles and a larger memory footprint as a result of more Java objects that get created).

EMN stores client profiles for each service it provides. One of its options is the 'terminal state': it refers to the final message delivery state that should be reported by the gates. More precise state information (real time delivery status upon reaching device as compared to best effort "push and forget" into network) can be obtained and hence allow for better reporting. This also means that extensive state information is persisted for longer periods (both in gateways and servers) ultimately affecting overall performance.

In order to avoid deep copying of entire messages when they pass through the engine, we have implemented a collection of mechanisms to ensure that only the references of messages are passed from the ingress point all the way to the egress channels. Specifically, a packing mechanism exists while forwarding the same message destined for multiple recipients from the server to the various egress channels. This can be viewed as a result of the extent of personalization required. If no personalization is used one version of the message can suffice for each type of channel which allows for an enhanced level of packing while forwarding messages from the server to the gates. Full personalization occurs in which each endpoint gets its own version of the message and therefore has a higher workload on the servers as it has to treat each message independently.

In addition, the submitted alert is treated as a stream, compressed at the ingress gate and transferred to the server for post processing. Further optimization routines are being investigated like binary XML. Transmission capacity is usually more limited and the loudest calls for binary XML have been among those using XML for message transport formats, including Web services users. One approach to achieving XML compression is to adopt a format that is designed for binary formats from the start. The leading candidate is ISO/ITU ASN.1, a data transmission standard that predates XML. ASN.1 is being updated with several XML-related capabilities that allow XML formats to be reformulated into specialized forms such as ASN.1.

Another optimization is the 'on-the-fly' compressing/decompressing of the SOAP requests. A request never exists in the full inside EMN: it is compressed as it arrives in the HTTP gate and parsed (into Java objects) as it decompresses in the server.

6 Conclusions

This paper started off by describing a mobile middleware platform, the AT&T Enterprise Messaging Network and an alert dissemination service that uses the platform. The contributing factors that go towards constructing a high performance alert dissemination engine were discussed. Specifically, performance testing was based on individual component benchmarking and an end to end benchmarking to determine overall system capacity. This helped to quantify system scalability parameters with respect to a particular hardware configuration. An alert splitting exercise was also shown to be promising to handle a very large alert efficiently. Splitting an alert improves the alert processing and dissemination time as a result of better resource management. Similarly the benefit of splitting an alert into smaller segments reduce when the number of segment increases. Towards the end we provide insightful comments and observations that should be used as a rule of thumb while constructing such an engine.

Acknowledgments

We would like to acknowledge the AT&T Research Real Time Customer and Service Management team and AT&T Government Solutions for their valuable efforts in deploying such a service and also without whom this would not have been feasible. We would also like to thank Kai Wei, who wrote some Java code for EMN performance testing while he worked as a summer student in AT&T.

References

[1] Y. Chen, H. Huang, R. Jana, T. Jim, M. Hiltunen, R. Muthumanickam, S. John, S. Jora, and B. Wei. imobile ee - an enterprise mobile service platform. *ACM Journal on Wireless Networks*, 9(4):283–297, July 2003.

[2] IETF. Rfc 1861 - simple network paging protocol. `http://www.ietf.org/rfc/rfc1861.txt?number=1861/`.

[3] S. Jora, R. Jana, Y. F. Chen, M. Hiltunen, T. Jim, H. Huang, K. Ow, A. K. Singh, and R. Muthumanickam. An alerting and notification service on the AT&T Enterprise Messaging Network. In *Proceedings of the IASTED Internet and Multimedia conference*, Grindelwald, Switzerland, February 2005.

[4] Sun Microsystems. Java message service. `http://java.sun.com/products/jms/`.

[5] AT&T Government Solutions. Chain-emn:. `http://www.att.com/gov/chain.html`.

[6] Inc. VoiceGenie Technologies. Voicegenie. `http://www.voicegenie.com/`.

[7] Wensong Zhang and Wenzhuo Zhang. Linux Virtual Server Clusters: Build highly-scalable and highly-available network services at low cost. *Linux Magazine*, November 2003.