

Adaptive Batching for Replicated Servers*

Roy Friedman
Computer Science Department
Technion - Israel Institute of Technology
Haifa, 32000
Israel

Email: roy@cs.technion.ac.il

Erez Hadad[†]
Distributed Computing Systems Group
IBM Haifa Research Lab
Haifa, 31905
Israel

Email: erezh@il.ibm.com

Abstract

This paper presents two novel generic adaptive batching schemes for replicated servers. Both schemes are oblivious to the underlying communication protocols. Our novel schemes adapt their batching levels automatically and immediately according to the current communication load. This is done without any explicit monitoring or calibration of the system. Additionally, the paper includes a detailed performance evaluation.

1 Introduction

Much of the overhead involved in sending a message is unrelated to the message's size. Examples include, e.g., the network (and middleware) protocol stack headers, the cost of invoking and handling system calls, and obtaining access to the network at the physical level. Also, there is the computational cost of handling messages both at the sender and the receiver in each of the MAC, networking, transport, session, and application layers. Taken separately, each of these costs may seem minuscule. Yet, together, they amount to a non-negligible overhead. Moreover, even when the impact of these overheads on latency seem acceptable, when it comes to throughput, they have a detrimental affect on the performance of the system. Specifically, if the computational cost of handling a single message is 1 ms, then the system cannot push more than 1,000 messages

*This research was partially supported by an IBM Faculty Award. Most of the hardware used for the performance measurements was donated to our lab by IBM.

[†]Major parts of this work were done while Erez was a graduate student at the Technion.

per second (on a single core machine). Similarly, if an application generates 50 bytes messages and the total headers overhead is 100 bytes, then at least two thirds of the network bandwidth is wasted on headers rather than on application's data.

Batching several small messages into a single larger message reduces the amortized per-message overhead, thereby boosting the throughput of the system. Consequently, this technique has become quite common in network-based applications [3, 4, 7, 8, 9, 12, 22, 23].

An important question in applying batching is how to decide when to generate a batched message. The trade-offs include balancing the throughput gain with the latency of a single message and obtaining good network and CPU utilization. That is, in order to increase the throughput, we may wish to pack a large number of messages in each batch. However, in order to obtain a large batch, we need to delay the first message for a considerable amount of time. Similarly, if the application generates messages slowly, then while the batching mechanism is waiting for messages to accumulate, the network and the CPU might remain idle.

Some batching schemes are *generic* in the sense that they do not take into account the specific characteristics of the underlying communication protocols nor the application. Other schemes are *specific*, meaning that their decision on when to generate a batched message depends on having intimate knowledge of their operating environment. Clearly, a benefit of generic schemes is that they provide better modularity and better separation of concerns, as called for by modern design principles. However, for this reason, generic schemes may not be able to achieve the same results as specific schemes.

Largely speaking, existing generic batching schemes can be categorized as *count-based* vs. *time-based*. In

count-based schemes, a batched message is sent only after a given number of application messages have been accumulated (or the total size of accumulated messages reaches some limit). On the other hand, in time-based schemes, a batched message is generated periodically, packing together all messages that have been accumulated since the previous batch was sent.

A problem with count-based schemes is that when the application is generating messages sparsely, they do not increase the throughput, since the system has enough CPU and network bandwidth to handle the load. Worse, since messages are arriving slowly, the latency hit caused by such batching becomes very high. In fact, termination is not guaranteed if an application generates fewer messages than the counter threshold.

Time-based schemes suffer from similar problems. If the batching timeout is too high, then the latency increase also becomes high, and when the application communicates sparsely, this does not contribute to the throughput of the system. On the other hand, if the timeout is set too low, then very little batching is done.

Of course, it is possible to combine a count-based scheme with a time-based one, by sending a batched message either if a given number of messages have accumulated, or a given timeout has past, whichever happens first. However, this still does not entirely solve the problems discussed above. The obvious remedy is, therefore, to incorporate some dynamic adaptivity to the batching parameters based on the actual communication pattern. Indeed, some works have suggested adaptive batching, by having a component that continuously monitors the communication pattern of the system and adapts the count threshold and batching timer to match the observed behavior using some heuristic [4].

There are several problems with the monitoring solution: First, monitoring adds an overhead to the system. Second, such a monitor sets the batching parameters based on the past, and so there is a delay between a change in the communication pattern and the reaction of the batching mechanism. This is especially bad if the communication pattern changes frequently, in which case a monitoring based solution will always be running one phase behind the actual communication pattern.

In this work we present two novel generic dynamic adaptive batching mechanisms, nicknamed *adaptive batching* (AB) and *timed adaptive batching* (TAB) that automatically and immediately adapt their batching level to the communication pattern, without any monitoring of the system. These schemes are generic in the sense that they do not assume anything about the underlying communication protocols, nor about the fre-

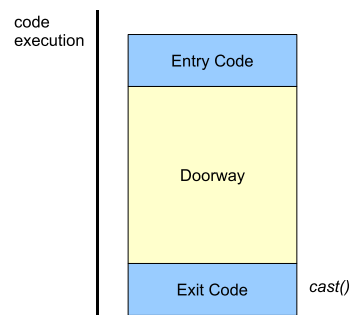


Figure 1. Adaptive Batching

quency in which the application generates messages. However, both schemes target replicated servers (or similar distributed applications) in which each server must broadcast each request to all other servers in the replication cluster. Moreover, for AB we also assume that overlapping requests are handled by multiple concurrent threads (yet, this is not needed in the case of TAB). We then present a detailed performance evaluation of both schemes, and compare them to Count-Based and Time-Based batching, in the context of FTS, a replicated server solution we have developed [15].

2 Adaptive Batching

Our adaptive batching schemes are targeting replicated servers and similar distributed applications. Generally, we assume a model where there are requests that need to undergo broadcast. Each request is handled by a separate thread. Each thread first executes some code, which possibly creates or prepares the request, and then invokes `cast()`, which is a primitive that adds the request to the current batch. Following the batching mechanism specifics, a batch-send event is generated from time to time, which triggers the sending (and resetting) of the current batch. As we later explain, the assumption of a multi-threaded application is required only for our AB scheme but not for our TAB scheme, which is completely generic.

2.1 Simple Adaptive Batching

The *adaptive*¹ *batching* (AB) mechanism, as shown in Figure 1, operates as following: each thread executes a common code path before invoking `cast()`. A part

¹The term “adaptive” has a different meaning when applied to distributed algorithms in general, such as adaptive mutual exclusion [2]. We chose to use this term, though, for the literal meaning of the word.

of this code path ending at the send message primitive is set as a *doorway*². The entry code in the beginning of the doorway is used for registering the thread executing it as being inside the doorway, e.g., by incrementing the doorway’s counter. Similarly, the exit code unregisters the executing thread, e.g., by decrementing the doorway’s counter. AB utilizes a doorway as following: the doorway entry code is inserted at some point that precedes the `cast()` invocation. The `cast()` method itself serves as the exit code. If a thread executing `cast()`, after un-registering itself and adding its request to the current batch, finds that no other thread is in the doorway, then it triggers the batch send event.

The intuition behind AB is that multiple requests should be included in the same batch only if they arrive “close together” to the `cast()` primitive, in the sense that the threads carrying these requests closely follow each other in invoking `cast()`. The doorway serves as a natural proximity sensor: the time required for a thread to cross it is a threshold. Should no other threads enter the doorway by that time, the batch is complete and sent, since the next threads’ requests are too far behind to join the requests in the current batch.

The length of the doorway code can be used to control the *sensitivity* of the mechanism: longer doorways cause long batches to begin forming at lower client loads than shorter doorways, since it becomes more likely that another thread would enter the doorway before the current thread exits it. Note, however, that the adaptive nature of the mechanism remains for every doorway length, as is later demonstrated.

2.2 Timed-Adaptive Batching

AB suffers from several shortcomings. First, it is not trivial to find a common code-path to serve as a doorway. The absence of a long-enough common code-path can be overcome by deploying multiple copies of the same entry/exit mechanisms along different code-paths. However, AB still requires careful code analysis (to ensure that threads actually pass through the entry/exit codes) and code modification. Next, the length of the AB doorway is set at coarse instruction granularity, making it hard to clearly evaluate the length since it is both code- and platform-dependent. Also, the possible doorway length is bounded by available code-path lengths. Last, implementing AB clearly requires a multi-threaded processing model, as the doorway works by detecting advancements of multiple independent execution contexts.

²The term “doorway” is used here in the same meaning as, e.g., in Lamport’s Bakery algorithm [19], as it operates on the same concept of a code segment that registers the threads inside it.

All the above problems are overcome by another AB variant named *timed adaptive batching* (TAB). The key difference between AB and TAB is that TAB employs a timer to serve as a doorway instead of a code-path. The timer is reset each time a thread wishes to invoke `cast()` (recall that the application level `cast()` method is only an indication that a thread wishes to broadcast a message and not the actual sending of the message). If the timer reaches a timeout, then the batch is actually sent (this is analogous in AB to a thread exiting the doorway with no other thread inside).

Interestingly, AB allows an immediate send of batch after the last thread of the current batch arrives with no other thread in the doorway. In TAB, on the other hand, the timeout still has to elapse after the last thread, thereby forcing a minimum non-zero additional latency even in the case of a single request in a batch. However, the TAB timer allows to define the doorway length in time units regardless of the implementation. Last, unlike AB, a timer-based implementation can also be used in a single-threaded event-based environment.

Notice that TAB is also significantly different from TB, although both mechanisms use a timer to trigger sending of a batch. TB’s timer triggers batch send at fixed intervals, regardless of the arrival rate of the batched messages. This guarantees progress of the message delivery but also introduces the aforementioned rigidity of the batching mechanism. TAB’s timer, on the other hand, triggers batch send only when messages are spaced sufficiently far apart, according to the doorway length parameter. Thus, TAB adapts its batching behavior to the message rate, but also only provides probabilistic progress guarantees, as discussed in Section 6.

3 Experimental Setting

3.1 Replicated Server Implementation

We have tested all batching schemes on an implementation of a replicated CORBA server called FTS [15]. FTS is entirely written in Java, and operates according to *active replication*, or *replicated state machine* [25], with a slight scalability twist: With FTS, a client connects to a single server using CORBA’s standard IIOP (over TCP/IP) connection. Whenever a server receives a request, it first broadcast the request to all other replicas in total ordering (ABCAST), by relying on an underlying group communication toolkit [6]. Once the request is delivered from the group communication toolkit, it is locally executed by each of the servers (under the assumption of deterministic methods). Only the server that

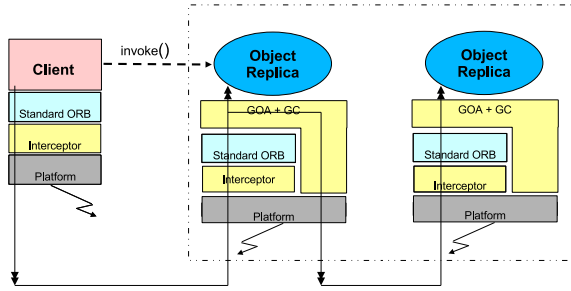


Figure 2. FTS Architecture

holds the IIOP connection with the client sends the reply as a standard IIOP reply. All servers, however, keep the reply in a local cache in case the request is later resent by the client, e.g., if the client timed out on the reply.

FTS is designed to work with any virtually synchronous group communication system using a generic Group Communication Interface layer. In the measurements reported in this paper, we have used Ensemble [16], which runs over UDP and is known to offer very good performance in LAN based clusters [10, 16].

If the connection between a client and a server breaks, or a request is timed out, this is caught by a *Portable Interceptor* at the client side [5, 11, 14, 24], which automatically redirects the request to an available server. The exact details of such client redirections are beyond the scope and interest of this paper (here we are interested in the benefit of batching), and can be found in [15]. The significant information is that this is done completely transparently to the client's code, which can be any unmodified CORBA client application.

Due to various CORBA limitations, and in order to remain transparent to the server's code, the logic for handling client requests at the server side is implemented inside a *Group Object Adaptor* (GOA), as illustrated in Figure 2. An *object adaptor* is the CORBA entity that mediates between the request broker and the actual server implementation; it is responsible for locating the implementation of a service, loading the object and starting it if needed, and for invoking the actual request on the object's implementation. In FTS we have extended the standard *Portable Object Adaptor* to include the replication and consistency logic, to form a GOA. The exact details are very technical, and the reader is referred to [15] for more information.

The important aspect of the above design for this paper is that in all our measurements, the batching mechanisms were implemented inside the GOA, and in particular between the application's code and the group com-

munication system. Thus, we only measure the performance of generic batching schemes. Also, request processing in FTS follows the model described in Section 2. Each GOA is equipped with a pool of threads (called the foreground pool) for handling client requests. Each of these threads processes the request it is assigned to and invokes `cast()` for batching-enhanced ABCAST of the request. Then, the thread waits for the request to be delivered back and executed, after which it returns a reply to the client. An additional background thread pool executes the requests delivered back from the ABCAST transport. In typical scenarios where the execution time is smaller than the combined latency of ABCAST and batching, we set the background pool to a much smaller size than the foreground pool in order to avoid idle processor due to exhausted foreground pool, as is demonstrated in Section 3.2 below.

3.2 Testing Environment

FTS was tested on an IBM JS20 Blade Center with 28 blades (machines). Each blade has two PowerPC 970 processors running at 2.2GHz with 4GByte RAM. The blades are divided into two 14-node pools, each internally connected with a Gbit Ethernet switch, and both switches are connected to each other. Additionally, the cluster includes an Intel X346 server with dual Pentium 4 Xeon 3.0GHz (hyper-threaded) with 2GByte RAM, used also as an NFS server for the common file system on which FTS is installed.

The blades and the server run Suse Linux Enterprise Server 9 (2.6.5 kernel) service pack 3. All FTS components run on IBM JVM v1.4.2 service pack 2 for PPC and Intel platforms, respectively. The ORB used is ORBacus for Java [17] v4.1.3. Ensemble [16] v2.01 is used for group communication.

In the tests, clients are running in blade pool 1 and servers in blade pool 2, so that all clients are equally distant from all the servers. Each client blade runs 250 clients sharing a common ORB. Each server blade runs a single server and a local Ensemble daemon. Last, the Intel server runs a *test manager*, whose purpose is to conduct a single test of clients vs. servers and an *executive* that generates sequences of tests, passes each test to the test manager and collects the results.

Each client runs as a separate thread performing a sequence of simple synchronous invocations on a separate server object. The interface of the server object is called `HelloObj` and consists of two methods called `set()` and `get()`. The `set()` method accepts a single string parameter and stores its value in the object, while the `get()` method returns the value of the stored string.

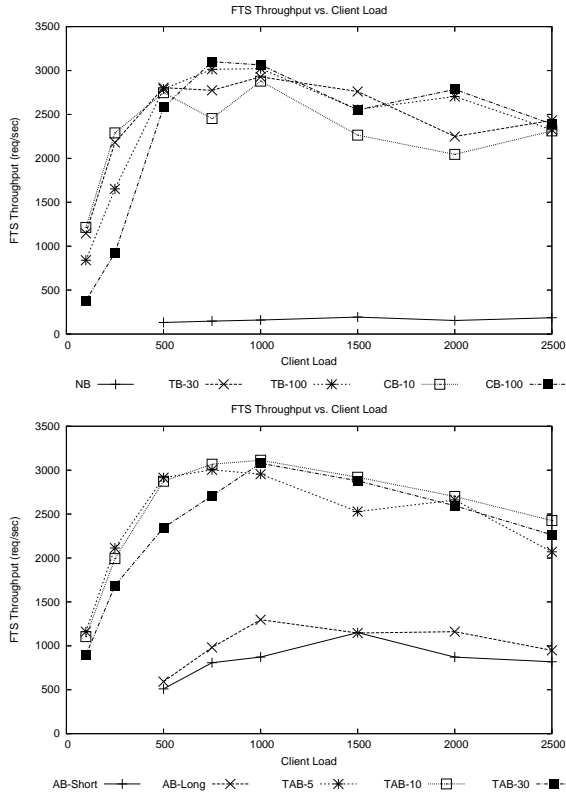


Figure 3. Throughput vs. Client Load (non-adaptive schemes on the left, adaptive on the right)

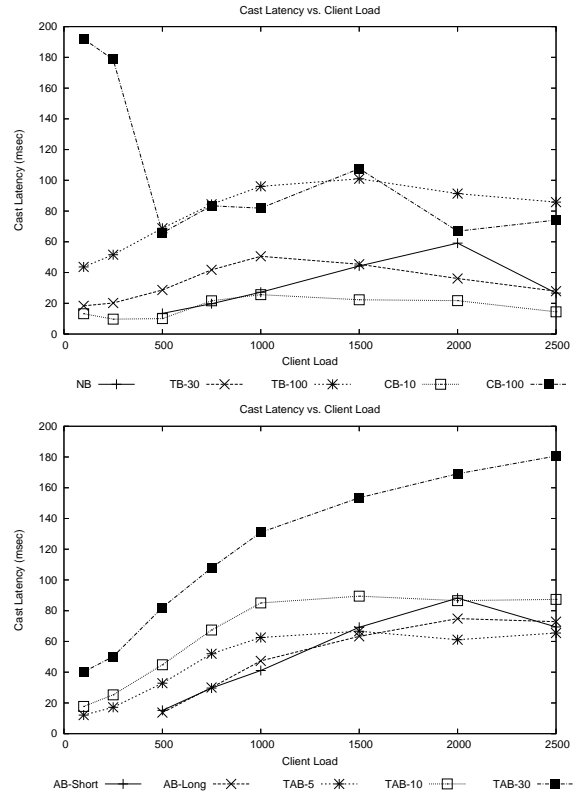


Figure 4. Cast Latency vs. Client Load (non-adaptive schemes on the left, adaptive on the right)

Each server has a 500-thread foreground pool for handling requests arriving from clients and a 20-thread background pool for executing ABCAST-delivered requests. The motivation for this allocation policy is briefly explained above in Section 3.1.

4 Performance Evaluation

To simulate varying client request rate, we used varying amounts of client loads, ranging from 100 clients to 2500 clients. The server cluster consists of 4 servers. For each test case, results are averaged over 4 runs. Each run consists of 2 phases of *warm-up* and *test*. During warm-up, each client performs a sequence of 500 requests. After finishing warm-up, a client notifies the test manager but continues sending batches of 100 “tail” requests, to maintain load on the servers. After the last client is done warming up, the test phase begins. Each client sends additional 500 requests as a test, notifies the

test manager upon completion, and continues to send “tail” requests, just as in the warm-up phase. Once the last client completes the test, the test phase ends. The test manager notifies all clients and servers to shut down and write the results. Measurements are collected only during the test phase, assuming that the servers are at a constant peak load during that time.

Following [13, 23], we used short requests in our tests, where the impact of batching should be most explicit³. However, since FTS operates at the CORBA application level, we cannot control the exact byte-size of a serialized request buffer. Also, note that the serialized buffer contains additional FTS data such as the request id. So, as a short invocation, we used the `set()` method of the target `HelloObj` object with a string parameter of 15 characters. Such an update request is very

³As shown in [13, 23], the effect of batching is smaller for long messages, since the further reduction of per-message overhead becomes marginal. See Section 5.

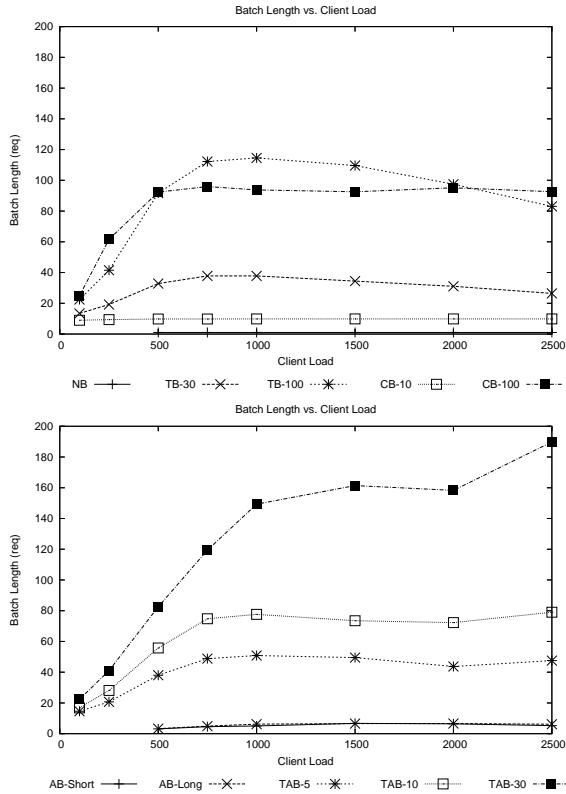


Figure 5. Batch Length vs. Client Load (non-adaptive schemes on the left, adaptive on the right)

small in CORBA terms but still translates to a buffer of 280 bytes. As we later show, batching still proves highly effective for requests of this size.

As for batching mechanisms, we used 5 different types. The first two are the classic methods mentioned before, namely Time-based Batching (TB) and Count-based Batching (CB). Next are the adaptive variants, code-oriented Adaptive Batching (AB) and Timed Adaptive Batching (TAB). Last, we also tested the case of No Batching (NB) as a base reference.

Figure 3 shows a comparison of the throughput of FTS with the various batching mechanisms, each with multiple values of parameters. Figure 4 compares the average per-request cast latency imposed by the batching mechanisms. Figure 5 shows the resulting average batch length for each of the mechanisms. Each mechanism's graph is marked by the corresponding acronym followed by the value of its parameter. For example: "TB-30" marks a test of the TB mechanism with a time

window size of 30 msec. Note that in each Figure, the graphs of the adaptive batching mechanisms are on the right pane, whereas the rest of the graphs are on the left pane, in order to avoid a crowded view.

The TB mechanism batches all requests accumulated over a predefined time window size. It is implemented by having a separate timer-thread wake up after each window elapses and send the currently accumulated batch. We tested a wide range of window sizes and present a few of the more characteristic exemplars, large window sizes vs. small window sizes.

The CB mechanism batches requests up to a predefined threshold. It is implemented simply by keeping count of the requests in the current batch and sending the batch after the last request is added. The backup timeout is implemented using a time-thread that is reset when the first request is added. For each of the CB thresholds demonstrated, we calibrated the backup timeout to be the lowest value we found that guarantees that the resulting batch size is very close (5% diversion) to the threshold at most of the client loads. As with the TB mechanism, we present the effects of both high and low counting thresholds.

The code-based AB variant was tested with doorway lengths that were determined according to the structure of FTS code. All the threads that handle requests in FTS execute the same code-path prior to invoking `cast()`. This code-path turns out to be quite short in terms of execution time. Thus, we defined only two lengths of doorway. The short doorway is implemented by putting the entry code just before the exit code, i.e., upon the invocation of the `cast()` method. The long doorway wraps the entire code-path by placing the entry code at its earliest beginning, which is the point where the client request first begins processing after being received.

Contrary to AB, the TAB variant's doorway length is defined in milliseconds. Note, however, that in the following graphs, the values used for testing are quite small, e.g., compared to the TB window size. This is because the doorway length determines a maximum time-distance between the arrivals of two consecutive requests, whereas the TB window size sets the entire interval required for the batch.

4.1 Discussion of the Results

Results Highlights: The bottom line of all our measurements is that TAB obtained good throughput, compared to all other schemes, under all client loads. In contrast, the non-adaptive schemes either performed well in high load, or in low load, depending on their thresh-

old/timeout. But none of the non-adaptive schemes performed well under all client loads. Yet, as expected, even the worst batching scheme is tremendously better than no batching at all. The bad surprise, for us, was AB, that obtained relatively poor performance. The reason is the lack of predictable long enough code paths. This indicates that TAB is the preferable scheme, since it assumes nothing about the application (other than the fact that the application generates messages), and consistently obtains very good results.

Detailed Analysis: Figure 3 clearly demonstrates the aforementioned rigidity of the classic CB mechanism. For low client loads, using a low threshold (10 request per batch) for CB yields good throughput in the sense that it is close to the maximum we were able to obtain for that load using any of the mechanisms and parameters. However, under high client load, throughput drops considerably. An opposite effect is seen when using high thresholds (100 requests per batch). The fixed threshold behavior is presented in Figure 5, showing the expected near-constant average batch length in all client loads except the very smallest. The average cast latency for CB tends to start high and decrease with the growing client load. This should be expected since as the client load grows, more CB batches are sent because of reaching the threshold rather than because of the timeout.

A similar rigid behavior is also seen in TB graphs. TB with a 100 msec window behaves similar to CB with a threshold of 100. Also, TB with a 30 msec window shows good throughput (same as e.g., TAB-5) in small client load but its throughput is definitely not the best at higher client loads. Also, TB batch length (Figure 5) grows with the client load. This is not surprising considering that the higher the client load, the more requests are likely to arrive during the same time interval.

Specifically, note that TB, when using small windows, seems to yield better throughput under high client loads than CB with small thresholds. The explanation for this lies in the implementation. The code inside `cast()` that adds a single thread's request to the current batch is defined as a mutually-exclusive critical section. Under high load, the timer-thread scheduling is often delayed because of the large number of concurrent threads. As a result, more requests are added to the batch than there should have been allowed for the given window size. The support for this explanation is given by Figure 4 showing that the average TB cast latency under high load is often higher than the window size. If the implementation did not suffer from delayed closure of batches, the cast latency should have remained very

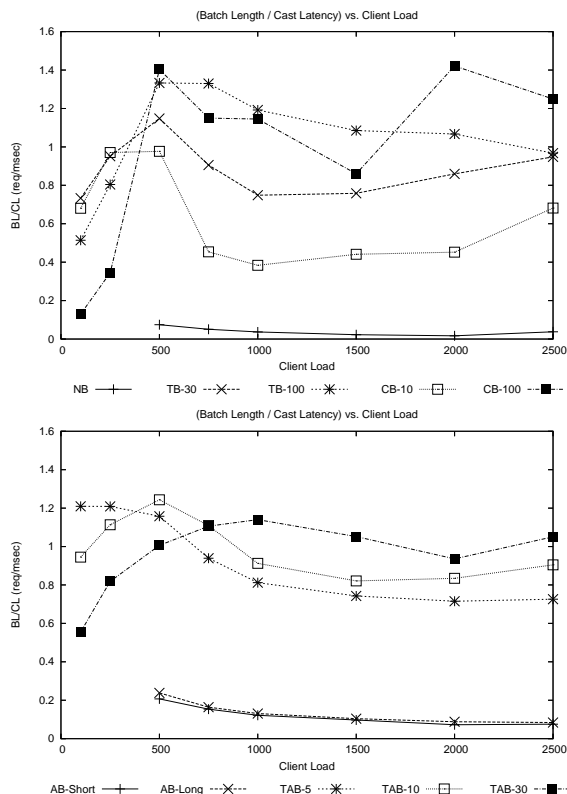


Figure 6. (Batch Length / Cast Latency) vs. Client Load

close to the window size.

Both AB and TAB show the expected adaptive behavior, as can be seen in Figures 4 and 5. Both batch length and average cast latency start low and grow with the client load. Yet, Figure 5 indicates the code-dependency limitation of AB. Profiling shows that the longest code-path doorway that could be defined is still very short in terms of computation time (less than 1 millisecond). Thus, the average batch length is small compared to the other mechanisms under the same load. TAB, on the other hand, as being able to work with much longer doorways, obtains much better throughput and truly proves the advantages of adaptivity, since it constantly maintains near-maximal throughput at all client loads.

Batch Length / Cast Latency: TAB's remarkable ability to maintain high (or even peak) throughput at all loads compared to the non-adaptive mechanisms can be explained through Figure 4.1, which plots the ratio of batch length over cast latency as a function of client load. Note that in this Figure, the behavior of each

batching mechanism's graph shows great similarity to the respective throughput graph of the same mechanism in Figure 3. This can be expected by considering the cast latency as an indication of the total batching latency, in the sense that a batch that takes a long time to construct, will most likely yield a high cast latency for its requests, and vice-versa. Therefore, a high value of the batch length to cast latency ratio will likely indicate that more requests are batched together in shorter time intervals, which results in a higher throughput.

As shown in Figure 4.1, all TAB graphs maintain high ratios at all loads. In contrast, the non-adaptive graphs tend to oscillate over a wide range of values, and specifically achieve low ratios at either low loads (for high thresholds, e.g., CB-100 and TB-100) or at high loads (for low thresholds, e.g., CB-10 and TB-30). The TAB graphs behavior results directly from TAB's tendency to batch together only requests that arrive close to each other, thereby achieving a high ratio value. In comparison, non-adaptive mechanisms tend to either include unnecessary delay in low loads or split bursts of close arrivals at high loads, which results in a low ratio. Therefore, Figure 4.1 presents adaptive behavior as a major contributor to obtaining good throughput under various client loads (or dynamic request arrival rates), as demonstrated in the TAB mechanism.

5 Related Work

An early classic example of batching (or packing) large amounts of information into a single packet can be found in the Nagle algorithm of TCP [22], where it is used as a means to improve a protocol's throughput by reducing its framing overhead, i.e., the ratio between its header size and payload size. More specific evidence of the advantages of packing together multiple messages into a single packet to significantly increase the throughput of an ABCAST protocol can be found in [13]. This work measured throughput and per-message latency of popular ABCAST protocols including the *rotating-token* (Rottoken) used by the Totem GC toolkit [21, 20] and the *dynamic-sequencer* (Dynseq) used by Amoeba [18]. Additionally, [13] also tested three variants of these two protocols, which were augmented with message-packing mechanisms.

The Rottoken protocol allows a process to order and send its pending messages only when it receives a token containing the ordering data that is rotated between the processes in a logical ring order, after which it passes on the token, piggybacked with the last pending messages or explicitly if no messages were pending. The

first packing variant of Rottoken (Denoted Rotfc) that tested in [13] simply packs the pending messages together upon token reception and piggybacks the token. Another semi-packing variant of Rottoken is *request-token* (Reqtoken) where the token passes upon request only between the actively-sending parties. Reqtoken packs only messages pending from before the token reception, so (typically few) additional messages will be sent unpacked until the token is passed on. Last, the Dynseq protocol orders messages by having the senders forward them to a *sequencer* process. Its packing variant (Dysfc) consisted of simple time-based accumulation of messages at the sender in a 1-msec window.

Tests of the protocols in [13] using bursty clients (single sender and all senders) and very short (1-byte) messages show a dramatic increase, by several orders of magnitude, of the throughput of the packing variants compared to all the non-packing protocols. The throughput increase is coupled with a significant decrease in average message latency, despite the delay imposed by the buffering of messages. These effects were attributed to the amortization of per-message overhead of headers and interrupt processing over multiple messages, as well as to reduced network contention due to the decreased number of on-the-wire messages.

Further support of the performance improvement of Totem using message batching is provided in [23]. In this work, Totem is augmented with a simple count-based message batching. That is, messages are added to a buffer until the accumulated buffer size surpasses a threshold, at which point the buffer is sent as a single packet. The resulting mechanism is both analyzed using stochastic tools and empirically tested, proving the increased throughput. Furthermore, this work checks performance under varying message size, concluding that the throughput gain of batching decreases when batching larger messages, up to a no-batching level, i.e., as if no batching is applied.

The count-based batching method is also explored in [9], combined with an *on-demand* protocol that resembles the Reqtoken protocol. The tests involved both a single and multiple senders with varying degrees of overlapping bursts and a fixed message size of 17 bytes, resulting in a 3- to 5-fold increase in throughput.

A recent paper [7] presents a general claim that increasing the *batching factor*, i.e., the number of messages-per-packet, contributes to increase the throughput of reliable multicast protocols, regardless of the batching method being used. The rationale behind this claim is that as networks bandwidth grows significantly larger (Gbit/sec), the bottlenecks of mes-

sage transmission shift from the network to the end-points processors, i.e., the senders and the receivers. Message batching reduces processing time at the end-points through overhead amortization, thereby increasing throughput. Using probabilistic analysis with conservative restraints of loss probability, the claim is proved for two simple types of unordered reliable multicast, based on positive and negative acknowledgments. Further empirical study of both protocols shows that the throughput indeed improves up to an order of magnitude as the batching factor grows.

One key factor that is missing from [7] is the message arrival rate. In other words, the model that is used for analyzing the throughput assumes that no time is wasted on waiting for arrivals of new messages to add to the current batch. As we show in our work, the latency imposed by the batching can actually lead to a much smaller increase of throughput, especially when the message arrival rate is low. The reason is, of course, the time overhead that is the sum of all the inter-batching delays together with the delay from the batching of the last arriving message to the batch send event. Thus, in order to achieve a large throughput increase, batching must be *tuned* or *adapted* according to the message rate.

The BFT work has reported an implementation of a Byzantine tolerant replicated NFS file server that obtains good performance [8]. A key optimization used in BFT is batching requests in the Byzantine consensus protocol that orders the requests. BFT uses a variant of count-based batching, in which messages are accumulated until their combined size reaches some limit.

A first-time notion of the need for adaptivity in batching mechanisms is presented in [4], using the Spread [1] GC toolkit and count-based batching. This work is focused on reaching and maintaining maximal throughput in a dynamic environment. First, an observation is made that throughput varies when using a constant batching threshold due to changes resulting both from porting the system to various platforms and from runtime fluctuations of message rate and size. Next, this work presents a simple rule of improving the throughput by constantly measuring the throughput and modifying the threshold accordingly. The result is close to peak throughput but has a tendency to oscillate due to temporal throughput measurement inaccuracy. Furthermore, the rule performs little modification steps and is likely to lock on to a local maximum of throughput rather than to advance to the globally-maximal throughput.

Last, the impact of batching is demonstrated in [3] when applied to a different context of implementing content-based publish/subscribe communications on top

of structured overlay networks. In general terms, this paper suggests extending the distributed hash table (DHT) function of the overlay to map both event-publications and event-subscriptions to partially-overlapping sets of overlay nodes so that for each subscription σ and event e that matches the subscription, there is at least one node r (designated as a *rendezvous* node) that will match e to σ and forward a notification by multicast to all the listed subscribers. By performing time-based batching of event notifications for the same subscription, the average number of hops required for event notifications is significantly reduced.

6 Conclusions

Our work is based on the observation that one of the key factors for obtaining good throughput under changing message rate is how well the batching is suited to the message rate so that it does not incur excessive batching latency. Consequently, we defined two adaptive batching mechanisms, code-based AB and time-based TAB. We showed the effectiveness of adaptivity by comparing the throughput of these two mechanisms with the classic time-based and count-based mechanisms, under a wide range of request arrival rates. Furthermore, the behavior of both adaptive mechanisms is controlled by a single parameter, the doorway length. Yet, adaptivity and the resulting good throughput are maintained for a large range of doorway lengths, making the mechanisms relatively easy to configure.

Both AB and TAB can also be combined with other batching mechanisms to achieve additional qualities. For example, in a model where the number of independent message arrivals is unbounded, neither AB nor TAB guarantee progress. In more common scenarios, neither of the adaptive variants provides *predictable* batching latency, so neither can provide bounded overall transmission latency even if the transport latency is bounded. Both issues stem from the fact that the batching latency depends on the temporal amount of closely following request arrivals. Progress can be guaranteed by adding a timeout or a threshold limitation, albeit this reintroduces rigidity into the algorithms, especially during high message rates.

Still, a timeout would also guarantee predictable batching latency. Furthermore, under high load, an adaptive mechanism combined with TB or CB can also guarantee predictable minimal throughput. For example, consider TAB with a doorway length of τ time units combined with TB with a window size of T time units. During high load, when the batch grows to cover the en-

time-window, the number of messages in each time window, W is such that $W \geq \frac{T}{\tau}$. Thus, the throughput is at least $\frac{W}{T} \geq \frac{1}{\tau}$. A similar computation can be done for CB+TAB, yielding the same result.

The benefit of batching in lossy networks, such as wireless networks, might be reduced. This is because batching may increase both the probability and cost of batched message loss, due to batch messages being typically long and contain many messages. This issue is left as an open problem.

References

- [1] Y. Amir and J.R. Stanton. The spread wide-area group communication system. Technical Report CNDS 98-4, Johns Hopkins University, 1998.
- [2] J.H. Anderson, Y.-J. Kim, and T. Herman. Shared-memory mutual exclusion: Major research trends since 1986. *Distributed Computing*, 16(2-3):75–110, 2003.
- [3] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 437–446, Washington, DC, USA, 2005. IEEE Computer Society.
- [4] A. Bartoli, C. Calabrese, M. Prica, E.A. Di Muro, and A. Montresor. Adaptive message packing for group communication systems. In *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*, volume 2889/2003 of *Lecture Notes in Computer Science*, pages 912–925. Springer-Verlag GmbH, 2003.
- [5] T. Bennani, L. Blain, L. Courtes, J.-C. Fabre, M.-O. Killijian, E. Marsden, and F. Taïani. Implementing simple replication protocols using corba portable interceptors and java serialization. In *International Conference on Dependable Systems and Networks (DSN)*. 2004.
- [6] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.
- [7] B. Carmeli, G. Gershinsky, A. Harpaz, N. Naaman, H. Nelken, J. Satran, and P. Vortman. High throughput reliable message dissemination. In *SAC: Proceedings of the ACM symposium on Applied computing*, pages 322–327, New York, NY, USA, 2004. ACM Press.
- [8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.
- [9] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed Systems Engineering*, 4(2):109–128, June 1997.
- [10] V. Drabkin, R. Friedman, and A. Kama. Practical byzantine group communication. In *Proceedings of the 26th International Conference on Distributed Computing Systems (ICDCS)*, Lisbon, Portugal, July 2006.
- [11] R. Friedman and E. Hadad. Client-side Enhancements using Portable Interceptors. *Computer System Science and Engineering*, 17(2):3–9, 2002.
- [12] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. In *Proc. of the 15th Symposium on Reliable Distributed Systems*, pages 140–149, 1996.
- [13] R. Friedman and R. van Renesse. Packing Messages as a Tool for Boosting the Performance of Total Ordering Protocols. In *Proc. of the Sixth IEEE HPDC*, 1997.
- [14] F.G.P Greve and J.-P. Le Narzul. Implementing ft-corba with portable interceptors: Lessons learned. In *Workshop on Fault-Tolerant Computing, in conjunction with SBRC 2002: Brazilian Symposium on Computer Networks*, May 2002.
- [15] E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical Report CS-2004-03, Technion, Israel Institute of Technology, 2004.
- [16] M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
- [17] IONA. IONA Technologies. <http://www.orbacus.com>.
- [18] M.F. Kaashoek and A.S. Tanenbaum K. Verstoep. Group communication in amoeba and its applications. *Distributed Systems Engineering Journal*, 1(1):48–58, 1993.
- [19] L. Lamport. A New Solution of Dijkstra’s Concurrent Programming Problem. *Communications of the ACM*, 17(8):453–455, August 1974.
- [20] L. Moser, P. M. Melliar-Smith, D. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4):54–63, April 1996.
- [21] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos, and T. P. Archambault. The Totem System. In *Proc. of the 25th Annual International Symposium on Fault-Tolerant Computing*, pages 61–66, Pasadena, CA, June 1995.
- [22] J. Nagle. Congestion control in ip/tcp internetworks. IETF Network Working Group, RFC 896, January 1984.
- [23] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Message packing as a performance enhancement strategy with application to the totem protocols. In *Proc. of GLOBECOMM '96*, pages 649–653. IEEE, 1996.
- [24] OMG. Portable Interceptors. ptc/01-03-04.
- [25] Fred B. Schneider. The state machine approach: a tutorial. Technical Report TR 86-800, Department of Computer Science, Cornell University, December 1986. Revised June 1987.