Using Selective Acknowledgements to Reduce the Memory Footprint of Replicated Services*

Roy Friedman[1] and Erez Hadad[2]

[1] Computer Science Department
Technion - Israel Institute of Technology
Haifa 32000, Israel
roy@cs.technion.ac.il

[2] IBM Haifa Research Labs
Haifa University
Mt. Carmel, Haifa 31905, Israel
erezh@il.ibm.com

Abstract. This paper proposes the use of *Selective Acknowledgements* (SACK) from clients to services as a method for reducing the memory footprint of replicated services. The paper discusses the general concept of SACK in replicated services and presents a specific implementation of SACK for an existing replication infrastructure. Performance measurements exhibiting the effectiveness of SACK are also presented.

1 Introduction

Context of this study: With our growing dependence as a society on computing infrastructure, fault-tolerance and high availability have become implicit requirements from any service. Yet, a grand challenge is how to provide the required levels of fault-tolerance and high availability at affordable costs. One of the main techniques for obtaining these goals is by using replication, as it allows harnessing together standard hardware and software components, thereby enjoying the economy of scale in terms of price/performance.

At the same time, such fault-tolerant implementations must take care to handle consistency and invocation semantics correctly. Specifically, in order to ensure the availability of the service to the client, a client may time out on a request to a server (or a set of servers), in which case the client will reissue the same request. Notice that such a time-out and reissuing of the request can be caused either due to real server failure, or due to networking problem. Either way, it is possible that the request has already been executed by the server, and it is only the reply that did not reach the client on time. Thus, in order to provide an *at-most-once* invocation semantics, it is important to be able to detect when the original request was already executed by some server. The server (or servers) receiving the reissued request must then return the reply that was

* 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.

generated for the original request and avoid re-executing it.[3] This is typically obtained through a *request/reply* cache (or a request log).

Hence, the request/reply cache (or request log) is a fundamental component of server replication [3, 19, 22] and also of other distributed services [24, 28] that need to maintain *at-most-once* request execution safety. The cache stores client requests received by the replicas of the service. When a client issues a request to a server, the server checks the request ID at the cache to determine whether it, or some other replica of the service, has already processed this request. If so, the server returns the matching stored reply instead of re-processing the request. This ensures that a request is processed at most once.

Due to the need to store all requests and replies, the cache is often one of the most memory-consuming elements of a replicated server. Excessive cache growth might eventually impede performance due to memory swapping and, in languages like Java, dynamic memory management overhead. Furthermore, once memory is exhausted, servers might crash and the service might be disrupted[4]. Hence, slimming down the cache may increase both the replicated service's performance and scalability. Of course, it is possible to store some of the cache's contents on disk in order to save memory, but this entails a significant latency.

In most client/server middleware standards, clients may perform multiple independent invocations on a server object using different threads or asynchronous invocations. As a result, a new request arriving from the same client machine cannot be assumed to implicitly acknowledge receiving replies for any previous requests that have finished processing. Furthermore, most middleware implementations do not provide the server with any indications of a successful delivery of replies to clients. Thus, the common method of eliminating old requests from the cache is based on a globally known time-limit after which the request is assumed not to be re-issued.

Several ad-hoc solutions might be considered for reducing the cache size. Shortening the timeout could be effective, yet below a certain point, it might compromise the ability to guarantee at-most-once semantics. Also, adding purging cycles might remove requests with greater time-out accuracy but would also require more CPU, which is likely to degrade performance without considerably reducing the cache since the time-out remains the limiting factor. Hence, we considered adding information to the purging process to make it possible to remove stored requests from the cache even before the timeout without violating safety conditions of at-most-once and at-least-once request execution.

---

[3] Some solutions involve the client joining a reliable multicast group with the server, typically implemented with a group communication toolkit [14, 17]. This can eliminate the need for reissuing requests. However, such approaches suffer from serious scalability problems; either a new group must be created for each client, or a new view must be generated for each client entering and leaving the system. Both options are very costly and do not scale well in the number of clients.

[4] In fact, the motivation for working on this problem began after noticing that the servers running a replication infrastructure we developed, called FTS, crash-fail under high loads due to excessive memory requirements.

Contribution of this work: In this work we propose to use a selective acknowledgement (SACK) mechanism in which clients notify servers about which replies they have already received. These acknowledgements are typically piggy-backed on existing messages, thereby rarely generating traffic of their own.

We also describe how SACK is implemented within a replication infrastructure we have developed, called FTS, for CORBA objects. In this implementation, SACK is completely transparent to the client's application, as is the rest of FTS.

Finally, we present a performance analysis that exhibit the benefits of using SACK. It shows a significant drop in the memory footprint, and even a slight improvement in throughput.

The rest of this paper is organized as follows: Section 2 presents the implementation of SACK, both as a general concept and the specific implementation in FTS. Section 3 includes the performance evaluation, and we compare SACK with related ideas in Section 4. Finally, we conclude with a discussion in Section 5.

2 Implementing SACK

2.1 Basic Idea

As discussed above, the basic idea of SACK is fairly simple: have each client send to the server(s) the list of requests ids for which this client has already received a reply. When the servers receive such a message, they can eliminate the corresponding requests and replies from their cache. However, there are several additional issues and pitfalls that need to be handled, as we discuss below.

Ideally, the SACK mechanism, like the rest of the fault-tolerant infrastructure, should be as transparent to the application as possible. Several middleware standards, such as CORBA and Microsoft .NET, offer standard interception mechanisms (Portable Interceptors in CORBA [23, 8] and Custom Sinks and Filters in .NET). These can be used to hide SACKs from the application's code. Below, we give a specific example of a CORBA implementation.

Another issue in replicated services is that the request and reply should be propagated to the caches of all servers. This is in order to handle a situation in which the re-issued request arrives at a different server. In active replication [25], where all servers execute the request, the request is propagated to all servers using some totally ordered delivery mechanism (ABCAST), e.g., [1, 4, 9, 16]. Each server executes the request after receiving it from the ABCAST. Thus, implicitly, each cache includes the request. However, in primary/backup replications styles [5], the request is executed by only one server, and an explicit propagation of the request and reply to the caches of all other replicas is needed.

Yet, if care is not taken, a race between a SACK and the propagation of the original request to all replicas could occur, which may cause some servers to either never execute a request, or to drop the SACK. For this reason, SACK propagation should be made consistent with the actual requests propagation. More details appear below in Section 2.4.

2.2 Overview of FTS

We have implemented the SACK mechanism inside a CORBA replication infrastructure called FTS [11]. 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 [5]. 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). After execution, only the server that 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. Specifically, in the measurements reported in this paper, we have used Ensemble [13], which runs over UDP and is known to offer very good performance in LAN based clusters [7, 13].

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 [2, 8, 10, 23], 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 [11]. The significant piece of 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 1. 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 [11] for more information.

The important aspect of the above design for this paper is that the SACK mechanism is implemented inside the Portable Interceptor (PI) on the client side and inside the GOA on the server side. The details appear below.

2.3 Implementing SACK in FTS

On the client side, SACKs are transparently handled by the FTS PI, which collects request IDs of replies and attaches them as SACKs to subsequent invocations that target the same object the replies were received from. This also ensures that SACKs of multiple independent invocations, such as by multiple client threads or using asynchronous invocations, are also collected and sent in a subsequent request from the same client to the same object. Thus, even if a client thread interacts with one service object and then

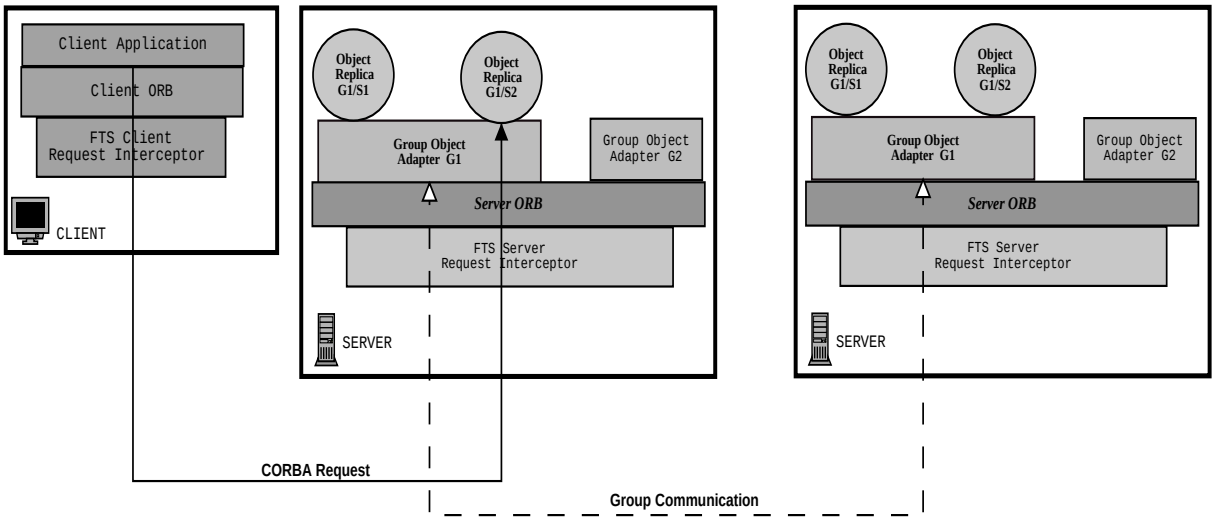

Fig. 1. FTS Architecture

with another, the SACKs that were collected from its invocations of the first object may still be sent along with a different thread's invocation.

Furthermore, FTS replicates multiple objects together in replication context groups. Accordingly, FTS also allows to accumulate SACKs according to the target replication context group (which may contain multiple objects) instead of only the target object in order to further increase SACK coverage. That is, if a thread invokes an object inside a particular context, the request carries SACKs of requests invoked by any thread at the same client that previously invoked an object in the same context.

On the server side, SACKs are handled inside a `SackHandler` object in each GOA. This object holds two SACK stores called `arrived` and `cast`. The `arrived` store contains SACKs that arrived from clients and are pending ABCAST. The `cast` store contains SACKs that were delivered back from ABCAST and need to be processed. The `SackHandler` object is operated by a dedicated daemon thread that periodically (once per second) collects and sends the `arrived` SACKs in batches, and then processes `cast` SACKs by locating their matching requests in the cache and tagging them as removable. The actual removal of a request takes place only after the request is done executing. After being processed, the `cast` SACKs are discarded. Further implementation details can be found in [12].

2.4 Rationale

In many common distributed applications, such as a database service, a directory service, etc., a client interacts with a service object using a sequence of invocations rather than a single invocation. Thus, subsequent invocations can be used by the client for acknowledging successfully-received replies of previous invocations. Consequently, on the server side, acknowledged requests can be discarded from the cache.

This concept resembles the technique used by TCP [27] and SACK TCP [18] to remove acknowledged segments from the sending window. However, unlike TCP or SACK TCP, a generic middleware SACK mechanism, such as inside FTS, cannot use the more space-efficient sequence ACKs, only selective acknowledgments (SACKs). The reason is that in many cases, request IDs are only guaranteed to be unique, but not necessarily ordered. For example, this is the case with request IDs of FT-CORBA's `FTRequestServiceContext` [22]. Of course, a specific implementation of FT-CORBA might provide ordered request IDs, but others may not. Hence, with the lack of guaranteed ordering among request IDs, acknowledging one request ID cannot be used to acknowledge additional request IDs that might have preceded it.

Sending SACKs through the ABCAST transport is justified as following. FTS does not assume clock synchronization between a client and any server, so the client PI does not expunge collected SACKs that have become invalid due to expired time-out at the servers. Therefore, a SACK that arrives at a server may refer to a request that is not in the cache, either since that request has not been delivered yet or since it has already been removed from the cache. With the server being unable to determine which case, the SACK is simply discarded with no additional effect. However, in order to ensure that a SACK is not discarded before the request it refers to is delivered to the server, SACKs are not processed upon reception from a client, but rather are sent over the inter-server ABCAST transport and used only when delivered back from it. Since the SACK, by definition, acknowledges a request that has already been delivered in at least one server, the total ordering property of ABCAST ensures that the SACK is delivered after the matching request in all the servers. This further ensures that all the servers receive the SACKs and use them to shrink their caches.

Note, however, that this method of SACK processing also results in an overhead in the form of ABCAST messages that are not used for relaying requests among the servers. This overhead is both in bandwidth and in message processing (sending and receiving messages), and is added to the processing overhead of the SACK daemon thread. The SACK message processing overhead is aggressively reduced by batching multiple SACKs together into SACK ABCAST messages, using simple time-based batching with a window size of 1 second[5]. This delay is still much smaller than the request expiration limit which is set to 10 seconds.

From an operational correctness point of view, SACKs are merely an optimization to the existing timeout-based purging mechanism. If a request has not been processed by a server yet, it may be tagged as removable by a SACK but it will not be removed from the cache until is is done processing. Also, clients send SACKs only after receiving a correct reply so a request removed through a SACK is guaranteed not to be needed again. Last, SACKs that refer to non-existing requests are simply discarded with no other effect. Therefore, these operating rules do not violate the underlying safety conditions of each request being processed at least once as well as at most once by each server.

[5] Actually, in our tests, SACK batches are also limited to 5000 SACKs each due to Ensemble daemon settings.

3 Performance Evaluation

We used FTS to test the performance impacts of the SACK implementation. The platform on which FTS was tested is 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 [15] v4.1.3. Ensemble [13] v2.01 is used for group communication.

FTS is deployed as following. 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.

We tested SACK under varying amounts of client loads, ranging from 500 clients to 2500 clients. The server cluster consists of 4 servers. We used short requests, where the overhead of ABCAST bandwidth should be close to maximum. The invocations are of the `set_greeting()` method of the target Hello object with a string parameter of 15 characters.

We conducted two simple sets of tests, one with SACKs and one without SACKs. The measurements that we used to compare the results are FTS throughput, memory usage and overhead estimates. Memory usage data consists of JVM allocated memory and cache size (counted in request records). This data is averaged over a sequence of read-outs performed once every two seconds during the test phase. SACK overhead is estimated as SACK percentage of total traffic. Thus, SACK bandwidth overhead is estimated in byte percentage and SACK message processing overhead is estimated in message percentage. The processing overhead of the SACK thread is harder to evaluate, so we use the throughput measurement as a global indicator for the entire processing overhead.

Figure 2 compares the cache sizes between runs with and without SACKs. Figure 3 shows a similar comparison of total server memory consumption. Note that the scale of 3 is in percents of the JVM memory allocation limit. The top allocation limit is the same in all runs (1GB) but is not relevant and therefore omitted. Next, Figure 4 shows FTS throughput with and without SACKs. Last, Table 1 shows the SACK overhead of ABCAST traffic in bytes and messages.

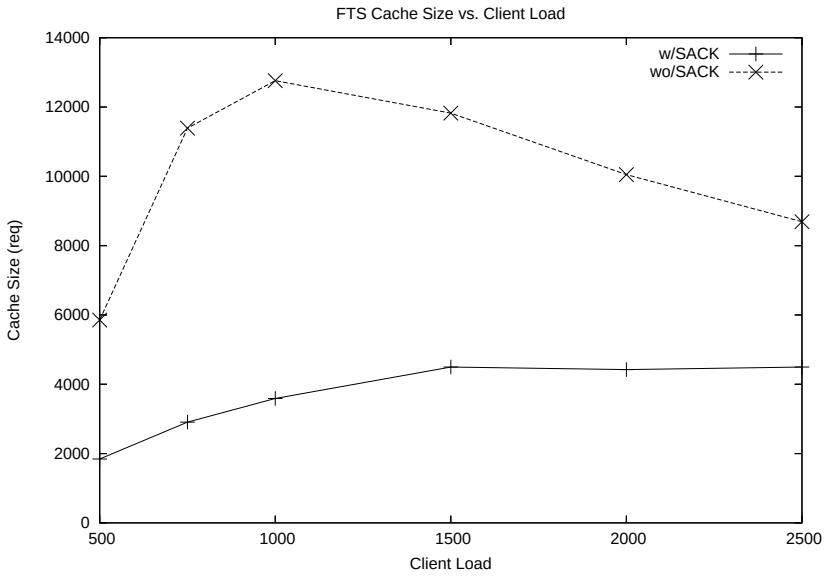

Fig. 2. Performance Evaluation: Cache Size vs. Client Load

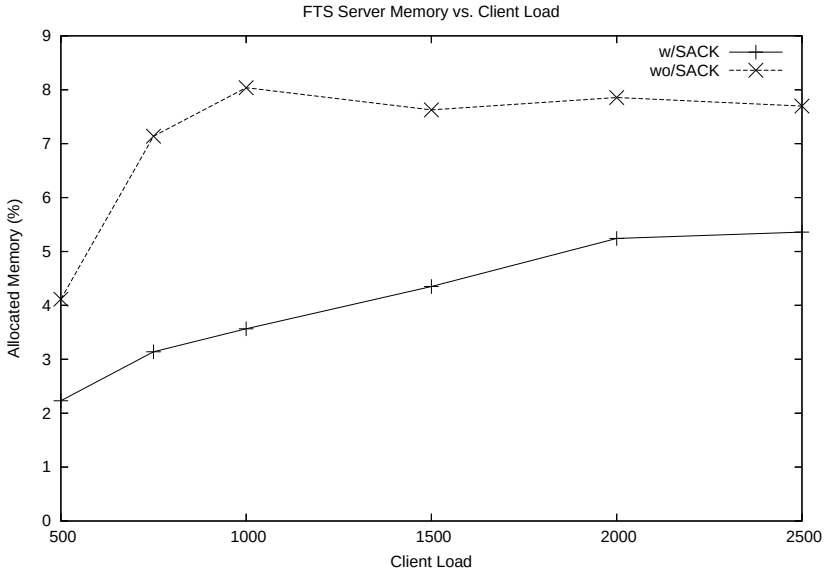

Fig. 3. Performance Evaluation: Server Memory vs. Client Load

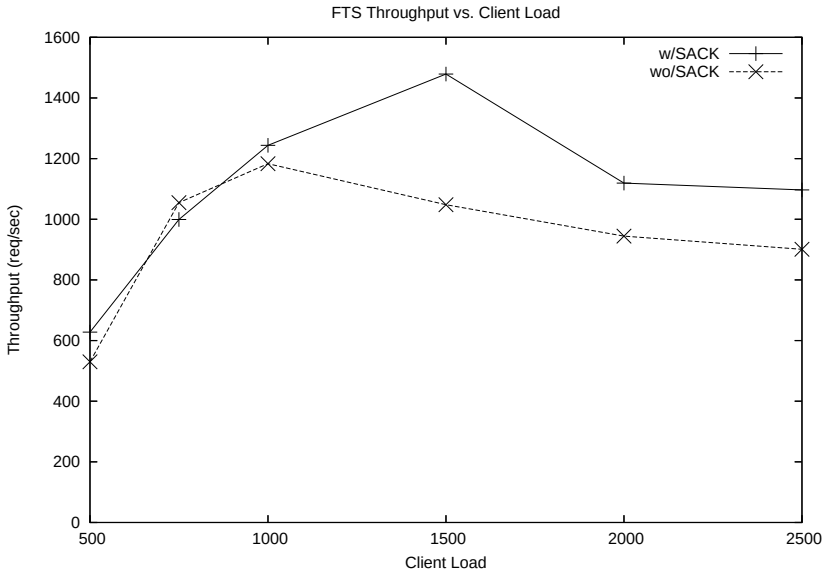

Fig. 4. Performance Evaluation: Cluster Throughput vs. Client Load

Client Load	Message Overhead (%)	Byte Overhead (%)
500	1.31	32.9
750	1.23	32.9
1000	1.29	32.9
1500	1.25	32.9
2000	1.11	32.9
2500	0.93	32.9

Table 1. Performance Evaluation: SACK Overhead vs. Client Load

3.1 Results Analysis

The results for the cache size, as shown in Figure 2 show that applying SACKs reduces the cache size by at least a factor of 2. In loads of 750 and 1000 clients the cache size is reduced by a factor of 3, which means that SACKs are indeed very effective.

An additional evidence of the positive impact of using SACKs is presented in Figure 4 that shows an improvement of about 20% in cluster throughput in high client loads. This means that despite the added processing and network overhead imposed by SACKs, the overall contribution improves not only server scalability but also performance. The throughput improvement can be mostly attributed to two factors. First, SACKs are handled in FTS by a dedicated thread that does not directly serve client requests. This may actually increase processor utilization by running in the idle periods in which all other threads are waiting for I/O operations. Second, the accelerated out-of-order removal of requests from the cache decreases the work of the cache purging thread, thus decreasing its contention with client threads that results from cache access synchronization and CPU utilization.

In the more general case (i.e., not necessarily FTS), any change to throughput following SACK usage may depend on the cache and SACK implementation. However, it seems reasonable to expect that applying SACKs would cause at worst a small decrease of throughput (due to the additional SACK processing), with each stored request being accessed and removed only once, either because of a SACK or because of a time-out. On the other hand, SACKs may also cause a positive throughput improvement through reduced cache contention and reduced memory swapping.

When comparing Figures 4 and 2, there are a couple of observations to be made. First, one can easily verify that the size of the cache when not using SACKs indeed matches the product of the cluster throughput and request expiration timeout (10 seconds). Second, the largest reductions of cache size occur at loads in which the cluster throughput is higher with and without SACKs compared to other loads. This can be explained as following. Each request occupies cache space as soon as it arrives at the server from the client/ABCAST transport, and stays there until being removed. Later, a SACK that matches the request arrives from the same client as that of the request. The higher the throughput, the lower the latency until that client sends the SACK attached to the next request. Thus, the time each request gets to stay in the cache is reduced with throughput increase.

Figure 3 clearly shows how applying SACKs reduces the overall memory consumption of the server by at least 50% for the client loads below 1000 and at all loads in no less than 40%. Also, following the previous observations, note that the larger memory reductions occur in the same client loads that exhibit higher throughputs with and without SACKs. This can be explained as following. The `SackHandler` causes removal of requests at the same rate as it accepts and processes SACKs, which matches the throughput. Therefore, the `SackHandler` on average contains as many SACKs as there are requests in the cache. Since each SACK consumes much less memory than a request (the size of only the request ID vs. the size of the entire request object), and the cache shrinks at least by a half (Figure 2), the overall effect is substantial memory reduction. Another observation arising from Figure 3 is the considerable weight of the

request/reply cache in the total server memory consumption, which is due to our test objects consuming little memory.

Last, we consider the message and bandwidth (byte) overhead as presented in Table 1. As this Table shows, the bandwidth overhead is fixed at approximately 33%. This is actually expected, since the number of SACKs received by each server is roughly the same as the number of requests. Thus, the ABCAST byte overhead roughly matches the ratio between the average size of a serialized SACK (approx. 90 bytes) and the average size of a serialized request (about 280 bytes). Note that since we explicitly used short requests in the test, the measured byte overhead is close to the maximum and will become much smaller when using typical requests with many more parameters and complex data types. The message overhead, on the other hand, grows smaller at higher loads. This can be understood when considering that SACKs are batched, so at higher client loads more SACKs get packed in fewer messages, yielding lower SACK message overhead. This also reduces the total processing overhead involved in SACKs, which can explain why the throughput improvement grows with client load as shown by Figure 4.

4 Related Work

Caches are employed in numerous types of applications. Typical examples can be found in areas of memory caching [29], web caching [30], file-system caching [24], database caching [26] and middleware object caching [6]. All of these cache implementations share a common quality of an *information proxy*. That is, the cache is used to store a copy of remote information (e.g., data, instructions, objects) in a location that is closer to the component that uses it than the source of the information, in order to improve access latency and/or throughput. In that sense, a cache may be considered an optimization that replaces direct access to the remote data, since it still maintains similar guarantees. Continued access to information (liveness) is ensured at the cache by techniques for retrieving remote data that are missing from the cache, e.g., cache miss and pre-fetching. Additionally, cache contents are kept correct w.r.t. the origin (safety) by having the cache guarantee a certain consistency condition, such as linearizability or sequential consistency [6].

In contrast, the request cache in fault-tolerant services is not an optimization at all. Rather, it is used to guarantee a safety condition of *at-most-once* semantics of request execution at the server. In other words, the cache ensures that the same request is not executed more than once, even if the client that sent it originally re-invokes the request due to not receiving the reply in the first time, e.g., following a temporal communication disconnection. In such cases, at-most-once semantics are maintained by having the client receive its reply from the cache instead of by re-executing the request, once the request is done executing for the first time.

A subsequent second difference between the request cache and typical caches is how the cache size can be limited. In typical caches the cache size can be arbitrarily limited by using various policies, e.g., LRU or LFU to enforce removing items when the cache becomes over-full. These policies do not harm the availability of the information since removed items can be retrieved back to the cache from the origin, e.g., when needed.

In fault-tolerant services, however, an update request that is removed from the cache without first ensuring that the client will not re-invoke it, may result in the request being re-executed, thereby violating the at-most-once safety condition. Thus, all update requests must be placed in the cache. A completed request can be removed only after ensuring that the client will not require the reply again. This may cause the cache to grow indefinitely depending on the cluster throughput and client load, instead of up to a fixed size.

As one can see from the text above, much more resemblance can be found between the request cache and the send-buffer management algorithm of reliable transport protocols, such as the sliding window of TCP [27]. Both the request cache and the send-buffer basically guarantee that the respective other side, namely the client in replicated services and the other peer of the TCP connection, eventually either receives the stored data or fails/disconnects. However, the request cache guarantees this property at a higher layer than transport, where a client may re-open a connection more than once in order to receive a reply to the same request. Thus, despite the typical transport layer of many middlewares being also TCP, the cache cannot rely on the guarantees of a single connection to deliver the reply. Furthermore, typically when working above TCP, there is no indication of successful delivery of the reply data to the client, and most middleware standards do not implement such a mechanism themselves. Consequently, one has to rely either on timing assumptions or on application-level acknowledgments in order to be able to safely discard the reply. TCP, on the other hand, uses timing assumptions only to detect whether the opposite peer has failed/disconnected and all of its packets are removed from the network. In contrast, TCP discards data from the send-buffer only upon receiving acknowledgments of successful delivery.

There are several other well-known distributed services that employ a server-side time-based request cache for the purpose of ensuring at-most-once request execution semantics, most notably the *Remote Procedure Call* (RPC) [28] and the RPC-based *Network File Service* NFS [24] services. However, to the best of our knowledge, no published works consider the scalability problem that could be imposed by such a cache and how to reduce such a problem using additional client acknowledgments.

The logging mechanism of FT-CORBA [22] operates quite similarly to a request cache. It stores completed requests and uses a time-out to discard them. Thus, applying our suggested improvement of adding SACK can contribute to reduce the log size of compatible implementations. Our specific implementation of SACKs in FTS where the receiving server propagates received SACKs to its peers using ABCAST to ensure causality (SACKs after matching requests) and to make SACKs effective at all the cache replicas, would also suit the warm-passive replication mode of FT-CORBA where a single primary updates the state of its backups. Active replication in FT-CORBA is less relevant since it assumes that a client receives its replies from all the servers using an ABCAST transport, in which case the log may not be needed for caching replies. One specific variant allows the client to communicate with the servers via a gateway, in which case the gateway should cache replies, and then SACK support can be applied to gateway replication.

However, not all FT-CORBA-compliant implementations use time-based logging. In Eternal [19, 21, 20], for example, a request/reply pair is removed from the log only

when the next request from the same client arrives. This is because Eternal assumes a certain behavior model for clients in which a client may issue at most one request at a time, sending another request only after receiving a reply to the first one. Thus, the arrival of a new request is an implicit acknowledgment of receiving reply for the previous request, which can now be safely discarded. This results in a cache that is only as large as the number of clients, but severely limits client behavior[6]. Contrary to this, by using SACK, there is no need to limit the client's behavior. In particular, an FTS client may invoke requests asynchronously or be multi-threaded. Consequently, requests can only be discarded based on explicit acknowledgments or time-outs. On the other hand, using time-outs impose stronger synchronization requirements in FTS (and in FT-CORBA) compared to Eternal, in the sense that there need to be known upper bounds on the times of request execution, client retry and client-server communication.

5 Conclusions

In this work we have shown the applicability of SACKs as an effective means to reduce the memory consumption of the request cache in replicated services, and, consequently, that of the servers running them. This work is based on an underlying observation that the request cache essentially resembles a sending window in sliding window protocols such as TCP, instead of a regular cache. Thus, it can use similar means, such as SACKs, for evacuating stored data.

After designing and testing the SACK implementation in FTS, we realize that SACKs are indeed a worthwhile modification. In addition to being a safe optimization, SACKs' incurred overhead is outweighed by their contribution to both server scalability and performance. Furthermore, the message and bandwidth overheads associated with SACKs are expected to grow smaller at higher client loads and with more typical usage that results in larger requests.

The proposed method of propagating SACKs to different cache replicas is a derivative of FTS architecture, where each server is responsible for relaying information received from its clients to other servers. Thus, it cannot be applied as-is to, e.g., FT-CORBA-compliant active replication, where a client communicates with all the servers through a proprietary protocol that guarantees request atomicity. Still, it is applicable to the warm replication style, where each primary server propagates client data/state updates to the backup servers. As a general concept, SACKs should be attempted as a means to improve scalability in various middleware implementations such as RPC [28] implementations that maintain at-most-once semantics by caching requests and using time-outs to purge the cache.

Last, SACKs may be especially applicable to 3-tier applications. In these applications, most of the state is stored in the 3rd layer (the database), making the reply cache of the 2nd layer a dominant memory consumer.

[6] In fact, without a backup evacuation timeout, the Eternal cache can overflow when many clients interact for short periods with the same service and then cease to. However, we found no documented evidence of such a timeout.

References

1. Y. Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, the Hebrew University of Jerusalem, 1995.
2. 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 *2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE, 2004.
3. A. Bhide, E. Elnozahy, and S. Morgan. A Highly Available Network File Server. In *Proc. of the USENIX Conference*, pages 199–205, 1991.
4. K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *Proc. of the 11th ACM Symp. on Operating Systems Principles*, pages 123–138, December 1987.
5. K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing Company and Prentice Hall, December 1996.
6. G. Chockler, D. Dolev, R. Friedman, and R. Vitenberg. CASCADE: CAching Service for CorbA Distributed objEcts. In *Proc. Middleware 2000: IFIP/ACM International Conference on Distributed Systems Platforms*, pages 1–23, April 2000. Best Paper Award.
7. 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. to appear.
8. R. Friedman and E. Hadad. Client-side Enhancements using Portable Interceptors. *Computer System Science and Engineering*, 17(2):3–9, 2002.
9. R. Friedman and A. Vaysburd. Fast Replicated State Machines Over Partitionable Networks. In *Proc. of the 16th Symposium on Reliable Distributed Systems*, October 1997.
10. 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.
11. E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical Report CS-2004-03, Technion, Israel Institute of Technology, 2004.
12. E. Hadad. FTS: A High-Performance CORBA Fault-Tolerance Service. Technical report, Technion, Israel Institute of Technology, 2006.
13. M. Hayden. The Ensemble System. Technical Report TR98-1662, Department of Computer Science, Cornell University, January 1998.
14. Isis Distributed Systems Inc. and IONA Technologies Limited. Orbix+ISIS Programmer's Guide.
15. IONA. IONA Technologies. http://www.orbacus.com.
16. L. Lamport. The Part-Time Parliament. *IEEE Transactions on Computer Systems*, 16(2):133–169, May 1998.
17. S. Landis and S. Maffeis. Building Reliable Distributed Systems with CORBA. *Theory and Practice of Object Systems*, April 1997.
18. M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. *TCP Selective Acknowledgement Options*. Network Working Group, April 1996. RFC 2018.
19. P. Narasimhan. *Transparent Fault Tolerance for CORBA*. PhD thesis, Dept. of Electrical and Computer Eng., University of California, Santa Barbara, December 1999.
20. P. Narasimhan and L.E. Moser P.M. Mellier-Smith. Strong Replica Consistency for Fault-Tolerant CORBA Applications. In *Sixth IEEE International Workshop on Object-oriented Real-time Dependable Systems*, pages 16–23, January 2001.
21. P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. In *Third USENIX Conference on Object-Oriented Technologies and Systems*, pages 81–90, June 1997.

22. OMG. Fault Tolerant CORBA specification, v1.0. ptc/00-04-04.
23. OMG. Portable Interceptors. ptc/01-03-04.
24. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. NFS version 3: Design and implementation. In *USENIX Summer*, pages 137–152, 1994.
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.
26. A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5 edition, 2005.
27. W. R. Stevens. *TCP/IP Illustrated, Volume 1*. Addison Wesley, October 1995.
28. W.R. Stevens, B. Fenner, and A.M. Rudoff. *Interprocess Communication*, volume 2 of *UNIX Network Programming*. Addison-Wesley Professional, November 2003.
29. A.S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall Inc., 4 edition, 2001.
30. D. Wessels. Squid web proxy cache.