FlexRPC: A Flexible Remote Procedure Call Facility for Modern Cluster File Systems

Sang-Hoon Kim, Youngjae Lee, Jin-Soo Kim

Division of Computer Science,
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon 305-701, South Korea
{sanghoon, yjlee}@camars.kaist.ac.kr, jinsoo@cs.kaist.ac.kr

***Abstract*— The concept of Remote Procedure Call (RPC) was proposed more than 30 years ago. Although various RPC systems have been studied and implemented, the existing RPC systems lack many crucial features and flexibility required for building modern cluster file systems.**

This paper presents *FlexRPC*, a flexible user-level RPC system that enables to develop high-performance cluster file systems easily. FlexRPC ensures client-side thread-safeness and fully supports multithreaded RPC servers. Parallel and serial multicasting mechanisms allow for implementing sophisticated replication in modern cluster file systems. The remote procedure can be invoked using both UDP and TCP transports with at-most-once semantics. The concurrent call requests are handled by a set of worker threads on the client and server side where the number of workers varies dynamically according to the request rate. In addition, the semantics and the specification of remote procedures are designed to be as close as possible to SunRPC.

The experimental results show that FlexRPC improves both latency and bandwidth significantly in spite of added functionalities. We also demonstrate the performance and the flexibility provided by FlexRPC by building working prototype of cluster file system called *Kadoop* on top of FlexRPC.

I. Introduction

Remote Procedure Call (RPC) is widely used to reduce the complexity and the development cost in building distributed systems. The goal of RPC is to make a remote procedure call as simple as any local procedure call. Remote procedure calls resemble local procedure calls both in syntax and in semantics so that the developer can easily invoke remote procedures over a variety of transports and physical networks.

The idea of Remote Procedure Call (RPC) has been discussed in the literature since at least as far back as 1976 [1]. The design possibilities have been examined in Nelson's doctoral dissertation [2] and Xerox developed several full-scale implementations such as Courier RPC [3] and Cedar RPC [4] in the early 1980's. The concept of RPC became popular as Sun Microsystems first implemented SunRPC on Unix platform. Since SunRPC (now called ONC RPC) specification was standardized as Internet RFC documents [5][6], various RPC mechanisms and implementations have been studied over the past 30 years to reduce the overhead [7][8] or to provide enriched calling semantics [9][10][11]. RPC has been the basis for building distributed systems, especially for many distributed file systems such as Sun's Network File System (NFS) [12], Andrew File System (AFS) [13], and Coda [14].

The current trend in building large-scale parallel or distributed file systems is to use a cluster of low-cost commodity hardware. For example, Lustre [15] is a cluster file system which aims at serving clusters with 10,000's of nodes, petabytes of storage, and 100's of GB/sec bandwidth. The Google File System (GFS) [16] is a scalable distributed file system for large distributed data-intensive applications. GFS consists of hundreds or even thousands of storage machines built from inexpensive commodity parts. The Hadoop Distributed File System (HDFS) [17] is a fault-tolerant scalable file system whose design is largely affected by GFS. HDFS is a distributed file system component of Hadoop, an open source project that supports distributed applications running on large clusters of commodity computers that process huge amounts of data.

While we develop our own large-scale cluster file system, we find that the existing RPC systems lack many crucial features. We have identified the following requirements that are considered necessary for implementing modern cluster file systems.

- **Client-side thread-safeness.**
  Due to the prevalent use of threads for lightweight concurrent computing, it should be allowed for multiple threads to invoke the same remote procedure on the client side without any problem.
- **Support for multithreaded RPC server.**
  The RPC server should be able to process the incoming requests concurrently using a number of server threads. The multithreaded RPC server not only achieves better throughput but also helps to exploit the current trend of using multiple processors or multi-core processors [20].
- **Support for various calling patterns.**
  Apart from the typical RPC model based on a simple request/reply pattern, modern cluster file systems also require more complicate calling patterns. As most cluster file systems rely on low-cost commodity hardware, component failures are the norm rather than the exception [16]. A common strategy to cope with such failures is to use replication, where the same file contents are replicated across multiple storage servers. A client may contact several storage servers to broadcast the file contents or to get the current state of each replica, whose

TABLE I
A COMPARISON OF KEY FUNCTIONALITIES IN VARIOUS RPC SYSTEMS.

RPC systems	Client-side thread-safeness	Multithreaded RPC Server	Other calling patterns supported	Semantics (on UDP)	Transports supported
Linux SunRPC [18]	Yes	No	–	none	TCP, UDP
Solaris SunRPC [19]	Yes	Yes	–	none	TCP, UDP
HP/Apollo RPC [11]	Yes	Yes	–	maybe, at-least-once, at-most-once	UDP
RPC2 [8]	No	Yes	parallel	at-most-once	UDP
ASTRA [10]	No	Yes	asynchronous	at-most-once	TCP, UDP
FlexRPC	Yes	Yes	parallel, pipelined	at-most-once	TCP, UDP

communication pattern should be handled efficiently by the underlying RPC layer.

- **Support for at-most-once semantics.**
  With the at-most-once semantics, the client retries the RPC call until it gets back a reply, and the server suppresses duplicated calls to make sure the call is not executed multiple times. It is desirable for RPC layer to provide the at-most-once semantics especially when some of remote procedures are not idempotent.
- **Support for various transports.**
  Two representative transport protocols in the Internet are UDP and TCP. Although RPC over UDP usually shows lower latency due to the lack of connection establishment, there is a limitation on the size of data that can be carried in a single UDP packet and no error handling is performed. Thus, UDP is suited to simple RPC calls with a small amount of argument or result. On the contrary, RPC over TCP exhibits better throughput especially when a large amount of data must be transferred. The diversity in the application characteristics necessitates simultaneous support for both UDP and TCP transports in the RPC system.

In this paper, we present a new RPC facility called *FlexRPC*. FlexRPC meets all the aforementioned requirements while providing greater flexibility for building modern cluster file systems. FlexRPC ensures client-side thread-safeness and fully supports multithreaded RPC servers. Parallel and serial multicasting mechanisms enable to implement sophisticated replication easily in modern cluster file systems. The same remote procedure can be invoked using both UDP and TCP transports with at-most-once semantics. In addition, the semantics and the specification of remote procedures in FlexRPC are designed to be as close as possible to SunRPC, which lessens learning cost and migration effort.

According to our experimental evaluations, FlexRPC shows significant improvement on latency and throughput in spite of added functionalities. The latency is reduced by up to 68% over UDP and by up to 79% over TCP compared to SunRPC. The effective bandwidth of parallel multicasting in FlexRPC has been improved by up to 79% compared to MultiRPC in RPC2. We also find that serial multicasting provides scalable bandwidth as the number of threads increases.

The rest of this paper is organized as follows. Section II reviews available RPC systems and their functionalities. Section III analyzes several issues related to the design of RPC system. Section IV presents the architecture and implementation details of FlexRPC. We will analyze the experimental results in Section V and draw a conclusion in Section VI.

II. RELATED WORK

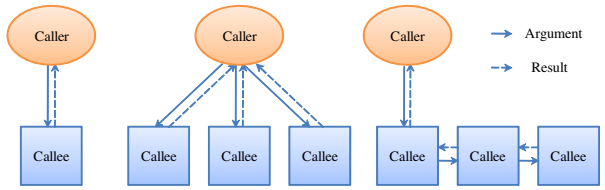
A survey on the early RPC systems has been conducted by Tay and Ananda [21][22]. SunRPC was introduced in 1981 by Sun Microsystems to extend the boundary of procedure call to remote hosts. It has been used as the basis for NFS [12] and now it is called ONC (Open Network Computing) RPC. The implementation of SunRPC was ported to various platforms including Solaris, Linux, and Windows. SunRPC uses portmapper that helps to locate service ports and batching RPC allows the caller to make several consecutive RPC requests without waiting for the response from the callee.

HP/Apollo RPC [11] was developed by HP/Apollo as part of the Network Computing Architecture. HP/Apollo RPC provides a rich set of RPC calls for programmers including a normal synchronous RPC, broadcast RPC, maybe RPC, and brodcast/maybe RPC. The maybe RPC does not expect any results back from server, and the last three types of RPC must be a request to an idempotent procedure. In addition, HP/Apollo RPC supports call abortion and callback mechanism which are found rarely from other RPC implementations.

Besides the traditional request/reply calling pattern, the need for supporting other RPC calling patterns has emerged during the development of various distributed file systems. CMU's RPC2 [8] supports MultiRPC which sends RPC requests to a group of callees in parallel. The receiver of a MultiRPC call cannot distinguish that call from a normal RPC as it is fully transparent to the callee.

ASTRA [10] exploits asynchronous RPC calling pattern. The request to the callee returns immediately and later the RPC system notifies the completion of the call to the caller by invoking the registered callback. The implementations of SFS [23], Chord [24], and LBFS [25] are all based on the similar asynchronous RPC mechanism that efficiently supports large numbers of simultaneous outstanding RPCs. Note that the functionality of MultiRPC can be also simulated by asynchronous RPCs.

GFS [16] and HDFS [17] replicate file contents among storage servers in a pipelined fashion in order to fully utilize each machine's network bandwidth and to minimize the latency to push through all the data. To the authors' best knowledge,



a) Single call b) Parallel multicasting c) Serial multicasting

Fig. 1. Calling patterns in RPC

there has not been any RPC system that supports this type of calling pattern.

Table I compares FlexRPC with the existing RPC systems. We can observe that none of the existing RPC systems meet our requirements for building modern cluster file systems described in Section I. It was frustrating for us to find that the current SunRPC implementation in Linux does not support multithreaded RPC server. RPC2 and ASTRA do not satisfy client-side thread-safeness, hence cannot be used with POSIX threads. Both Solaris SunRPC and HP/Apollo RPC provide limited support for calling patterns, calling semantics, or available transports.

III. Design Issues

A. Multithreading

Using multiple threads on the server side is an effective way in handling large numbers of concurrent requests from clients [26]. The SunRPC ported on Linux [18] stores socket descriptors that accept requests from callees in the thread-specific area. Consequently, assigning multiple threads to a service is structurally impossible and a service must be served by a single thread. The SunRPC implementation on Solaris [19] supports the multithreaded server model. Dynamically created threads share the socket descriptor set safely so that the server multiplexes and handles each service request independently.

Even though the callee is capable of servicing multiple call requests simultaneously, the RPC layer on the client side may not be thread-safe, i.e., only a single thread must invoke a call at any give time. Other caller threads should be serialized waiting for the completion of the pending RPC initiated by other thread, which incurs a long waiting time to make an RPC call.

The SunRPC implementations of Linux and Solaris are thread-safe. Both implementations guard shared variables with synchronization objects. RPC2 [8] makes use of its own LWP (lightweight process) package that allows to implement MultiRPC with multiple non-preemptive threads of control. The LWP package in RPC2 is, however, not compatible with POSIX threads (Pthreads) and RPC2 cannot be used for Pthreads-based applications. ASTRA [10] defines data structures used for receiving results as global variables without any protecting lock. Therefore, ASTRA is not thread-safe on the client side.

B. Calling Patterns

This paper primarily focuses on three types of calling patterns that arise during the development of modern cluster file systems.

A *single call* is the traditional request/reply-based calling pattern which is conceptually similar to a local procedure call. A caller requests a remote procedure to a single callee as shown in Figure 1(a).

When the same RPC request is delivered to multiple callees, we call it a *multicasting call*. In the multicasting call, the caller invokes a remote procedure with the same argument, and the call is completed if the caller receives the results from all the callees. We denote the number of destination callees as the *multicasting degree*. The need for multicasting calls arises in many cases. For example, the quorum consensus replication method [27] requires multiple sites to be contacted to perform an operation, and many P2P systems use multicasting calls to probe available nodes and files. Supporting multicasting calls in the RPC layer eases the implementation of such algorithms and systems.

The multicasting call can be divided into two categories: *parallel multicasting* and *serial multicasting*. In terms of calling semantics, parallel multicasting and serial multicasting are identical. The main difference comes from the way arguments and results are delivered, as illustrated in Figure 1(b) and (c).

In parallel multicasting, the caller delivers all the arguments to multiple callees at once and gathers the result from each callee to complete the request. On the other hand, in serial multicasting, the caller and callees cooperate to deliver the call request; the caller sends a request to one of the callees and the callee forwards the received request to the next callee forming a *delivery chain*. After the request is processed, each callee replies to the requester by adding its own result to the results obtained from the successive callee on the chain. Eventually, the caller receives the collected results from the first callee on the chain.

Parallel multicasting can shorten the latency at the expense of increased network bandwidth usage as the request is sent to multiple callees simultaneously. The network between the caller and callees can be easily saturated if the payload size or the multicasting degree increases. The Coda distributed file system [14] heavily relies on parallel multicasting via MultiRPC in order to maintain consistency among replicas.

Serial multicasting reduces the network traffic between the caller and callees by a factor of the multicasting degree since the caller transmits the request only once. As described in Section II, GFS [16] and HDFS [17] use serial multicasting to maximize write throughput while making a set of replicas. Since the propagation delay is short and the bandwidth is wide within a rack, the well-organized serial transfer achieves high throughput. However, serial multicasting incurs ambiguous semantics on partial success; the failure in the middle of delivery chain impedes the exact failure detection.

Asynchronous RPC provided by ASTRA [10] is another interesting calling pattern in which the caller does not wait for the completion of the request. Although asynchronous RPCs

can be used to implement parallel and serial multicasting, we do not see the support for asynchronous calling pattern is an essential feature for modern cluster file systems since it is difficult to use and error-prone due to the presence of a callback mechanism.

C. Network

To cope with different application requirements, the RPC system must be able to run over a variety of transport protocols. The representative transport protocols used in the Internet are UDP and TCP. UDP is a stateless, unreliable transport protocol where packets may arrive out of order, appear duplicated, or go missing without notice. UDP is meant to provide a low-overhead transport and useful for servers that answer small queries from huge numbers of clients. The maximum size of a UDP packet is limited to 64 KB including the header size. On the contrary, TCP guarantees reliable and in-order delivery of data from sender to receiver. In general, TCP incurs more overhead than UDP due to explicit connection establishment and added features such as flow control and congestion control.

If UDP is used, the system must handle or define semantics on packet loss. Since the packet loss is not reported to the sender nor the receiver, the system must detect the packet loss in some way. The widely used approach is to retransmit the same packet on timeout. If the call is idempotent or the application requires *at-least-once* semantics, the retransmission approach is enough. However, when the RPC layer wants to provide *at-most-once* or *exactly-once* semantics, the retransmission alone cannot be a solution to the packet loss problem. As Table I shows, SunRPC does not provide any mechanism for the packet loss under UDP. In this case, the application should implement its own module on top of the RPC layer that can guarantee the needed call semantics. Many other RPC systems such as RPC2 and ASTRA have the support for at-most-once semantics in which case the application can be simplified without paying attention to lost packets.

If TCP is used, the connection between the caller and the callee must be established before the actual communication. In SunRPC, the connection is established when an RPC handle is created, and it is closed when the handle is destroyed. Since opening a connection takes relatively long time due to 3-way handshaking, it is desirable to cache previous connections inside the RPC layer. The connection remains alive temporarily after the corresponding handle is destroyed, and it can be used for other RPC requiring the connection to the same callee.

IV. Architecture and Implementation of FlexRPC

This section explains the architecture and implementation details of FlexRPC. As shown in Figure 2, FlexRPC consists of four main modules: calling workers, handle manager, service workers, and response cache.

A. Calling Workers

FlexRPC uses multiple worker threads on the client side to invoke parallel multicasting calls. A parallel multicasting

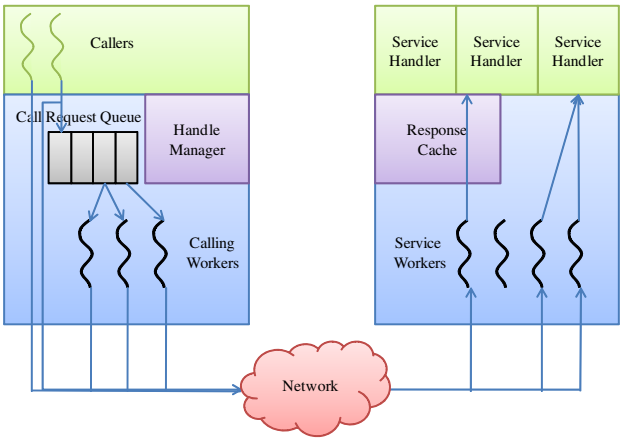


Fig. 2. FlexRPC system architecture

of degree n is divided into n single calls, called *subcalls*. The main caller thread puts the multicasting request of degree n into the *call request queue* if n is greater than one. The caller thread processes one of subcalls and the remaining $n-1$ subcalls are issued by other calling workers. After finishing the subcall, the caller thread checks the completion status of other subcalls. If there are pending subcalls that are not finished yet, the caller thread waits for the completion signal from the calling workers. When all the subcalls are completed or the completion signal is raised, the caller thread returns from FlexRPC with the results.

Calling workers monitor the call request queue waiting for any subcall request. If some requests are available in the queue, a calling worker increases the *issued* subcall counter and processes the subcall. Each subcall is handled as a normal single call where the issuing worker thread is blocked until the response arrives. After receiving the reply from the callee, the worker thread increases the *completed* subcall counter in the call request. The result is stored in pre-allocated area designated by the caller. When the completed subcall counter reaches the multicasting degree, the parallel multicasting call is finished and the last worker sends a completion signal to the caller thread.

The call request queue maintains a list of request entries. Each request entry contains the location of the buffer, the multicasting degree, the issued subcall counter, and the completed subcall counter. The calling worker which finishes the last subcall of the request dequeues the corresponding entry from the call request queue.

Insufficient number of calling workers will degrade system performance because some requests can be blocked due to the lack of available worker threads. On the other hand, the large number of calling workers will consume system resources too much. Hence, FlexRPC dynamically changes the number of calling workers on demand according to the request rate.

The calling workers are classified into active workers and dormant workers. Active workers represent the threads that are actively processing some subcall requests. The rest of the workers are called dormant workers. During inserting a request

entry to the call request queue, if the number of dormant workers is smaller than the multicasting degree minus one, the caller thread spawns more workers. On the other hand, if a worker thread remains as a dormant worker for a specified period of time, the thread terminates itself. This self-governing mechanism for calling workers avoids the lack of workers yet reducing the cost of keeping too many idle workers.

In FlexRPC, the calling workers are implemented with Pthreads. Hence, FlexRPC inherently guarantees thread-safeness on the client side and can be used with any Pthreads-based applications.

B. Handle Manager

Each call requires a socket (for UDP) or a connection (for TCP) to the callee. In FlexRPC, sockets and connections are encapsulated and stored in *handles*. The caller is not aware of any socket or connection. The caller simply acquires handles from FlexRPC and invokes one or more RPCs using the handles. And then the caller returns the handles back to the FlexRPC layer. Similarly, callees also use the handle notation for sockets or connections.

The *handle manager* is a module which manages sockets and connections. The handle manager reduces socket management overhead and connection establishment overhead by pooling a set of handles. It makes a decision on when a new connection is established or when the existing connection is closed.

Handles are cached into the *handle pool* which is indexed by the caller IP address and the port number. Upon the request for a handle from the caller, the handle manager looks up the handle pool to find the same handle created already. If there is such a handle, the handle manager checks out it from the handle pool and returns it to the caller. If there is no handle available, the handle manager creates a new handle by establishing a connection to the callee. If the caller returns the handle back, the handle manager inserts it into the handle pool.

Since the number of handles can be quite large, there is a configurable upper limit to the number of cached handles. If the number of cached handles reaches the limit, the handle manager picks up the least recently used handle and closes it.

Note that handles can become stale due to node crash or network problem while they are cached by the handle manager. The handle manager does not monitor the individual status of each handle, but invalidates related handles together when one of the existing handles becomes stale. During an RPC request, the FlexRPC layer communicates with the callee via the socket connection maintained inside the handle. If the connection terminates abnormally, FlexRPC sets a flag which indicates the handle is invalidated and returns an error code to the caller. Then the caller returns the handle back to the handle manager. When the handle is returned, the handle manager checks the flag. If the flag is set, all the handles in the handle pool connected to the same callee are invalidated, i.e., the handles are removed from the handle pool and destroyed. This prevents the caller from acquiring invalid handles. The handle manager cannot immediately invalidate active handles which are checked out from the handle pool and are being used by some applications. This normally does not pose any problem since the connection error will be propagated to the active handles with the same callee, and they eventually get returned to the handle manager.

C. Service Workers

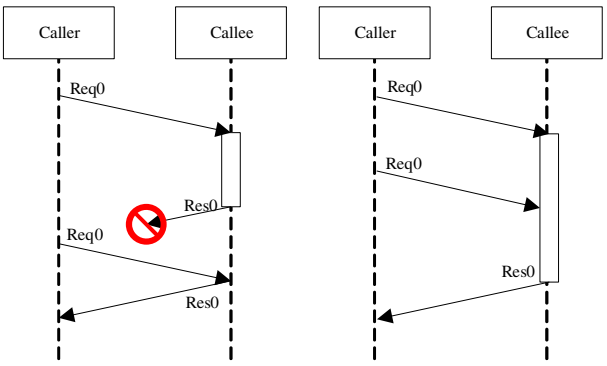
FlexRPC supports multithreaded RPC server using a set of *service workers*. The service worker monitor a set of socket descriptors using `select()` system call. The socket descriptor set is updated whenever a new RPC service is registered, a new caller connects to the server, or an existing connection is closed. If there is a call request, the service worker dispatchs the request and handles it. The incoming packet is analyzed and the designated procedure is executed by the service worker. Due to service workers, multiple call requests can be handled concurrently on the server side.

The number of service workers is adjusted dynamically similar to the calling workers on the client side (cf. Section IV-A). That is, the service workers are classified into active and dormant workers and a new worker is spawned when there is not enough dormant workers. The dormant worker is terminated after the specified timeout.

FlexRPC places a restriction on the payload size over UDP since the maximum packet size under UDP is limited to 64 KB. FlexRPC uses 40 bytes for its own header and another 8 bytes are used by UDP header. Thus, the maximum payload size for FlexRPC over UDP is 65,488 bytes. A single UDP request is received by `recv()` system call at once and it is processed immediately by the service worker.

For TCP, FlexRPC supports unlimited payload size. The payload over TCP can be divided into several packets. In that case, a partial payload is received by non-blocking I/O scheme. Since the TCP layer maintains the order of TCP packets, each packet is simply appended to the *request argument buffer*. The argument size is determined by the request header. If the whole argument is received, the service worker begins to process the call request. Since multiple RPC services can be registered, the service worker first finds the corresponding *service handler* of the request and then invokes the registered service handler with the received argument. As soon as the service handler finishes processing the request, the control comes back to the FlexRPC layer and the service worker sends the result back to the caller.

If the type of the request is serial multicasting, the payload must be forwarded to the successive callee on the delivery chain. The maximum length of the delivery chain is configurable and currently it is set to 16 by default. There are two forwarding strategies for serial multicasting. One is the *store-and-forward* scheme which forwards the call payload after receiving the whole payload from the sender. The other is the *pipelining* scheme which forwards the payload as soon as possible even if only a partial payload is available. The store-and-forward scheme can be easily implemented by invoking an RPC to the next callee with the received argument. However,



(a) Hit response cache (b) Ignore duplicated request on processing request

Fig. 3. Response cache on duplicated request

a series of calls is not overlapped since at any given time, only a single callee can forward the argument to another callee. On the contrary, the pipelining scheme can reduce the call latency by overlapping the payload transfer time among the caller and callees. With this reason, FlexRPC adopts the pipelining scheme in implementing serial multicasting. FlexRPC supports the pipelined serial multicasting for TCP only since it is originally designed to replicate a large file over TCP efficiently.

Since the packet can be corrupted during transit over the network, it is necessary to check the integrity of the received packet. In FlexRPC, the entire payload is verified by CRC32 checksum. There is a 32-bit checksum field in the header that is computed by the sender and checked by the receiver. If the mismatch is found on the checksum, the packet is recovered by requesting retransmission to the sender.

D. Response Cache

As we described in Section III-C, the simple retransmission strategy on packet loss merely provides at-least-once semantics while the support for at-most-once semantics is desirable. FlexRPC employs *response cache* to provide at-most-once semantics at the RPC level.

The response cache stores the recent results of the completed calls and detects duplicated requests. All the requests over UDP go through the response cache. The service worker asks the response cache to check the duplication of the request prior to processing of the incoming call. If the response cache finds the call is duplicated, it returns the result stored in the cache instead of executing the service handler again as shown in Figure 3(a). The service worker then sends the result back to the caller. In case the duplicated request is detected but the result is not ready meaning the call processing is still in progress, the response cache returns a special code that makes the service worker to ignore the request. This is not harmful as the result will be generated by the previous call and sent back to the caller eventually. Figure 3(b) illustrates this scenario.

The response cache detects the duplicated request with *call identifier*. Each call has its own unique call identifier that consists of caller IP address (`cid`), process ID (`pid`), handle ID (`hid`), and transaction ID (`tid`). The `cid` and `pid` enable multiple callers to coexist in a host. The `tid` is used to identify each request. The unique `tid` is issued on each call and the same `tid` is used during retransmission. Therefore, if a request uses the call identifier that is used already, it indicates the request is retransmitted. The `hid` helps to determine the completed requests so that the response cache keeps only the latest requests actually in progress. Since a handle is used by a dedicated caller at any given time and every RPC in FlexRPC is synchronous, the handle serves exactly one request at a time. Therefore, if the `tid` is changed for the given `hid`, it means the request with the previous `tid` has been completed and the cache entry of the old request can be removed from the response cache.

The recent requests are stored in the response cache indexed by the call identifier. The requests with the same `cid` an `pid` are clustered to check related requests fast. In order to minimize resource usage, the response cache stores only the result discarding the argument. In addition, the lifetime of each cache entry is checked periodically and obsoleted entries are removed from the response cache.

E. IDL and Stub Generator

Usually, the specification for remote procedures is described in Interface Definition Language (IDL). IDL focuses on the interface and the type of the argument and the result rather than the internal of the procedure. Converting an IDL file into the source files in the target language is automatically done by a stub generator. For example, SunRPC uses a stub generator called *rpcgen* which generates client-side stubs and server-side stubs with an appropriate header file from a specification written in *RPC Language*. These stubs are compiled and integrated with the application.

FlexRPC uses the IDL identical to the SunRPC's. As Table II shows, most of FlexRPC interfaces are named after SunRPC interfaces. Therefore, FlexRPC is straightforward to understand and easy to learn. FlexRPC's stub generator was also built by modifying SunRPC's rpcgen.

V. Evaluations

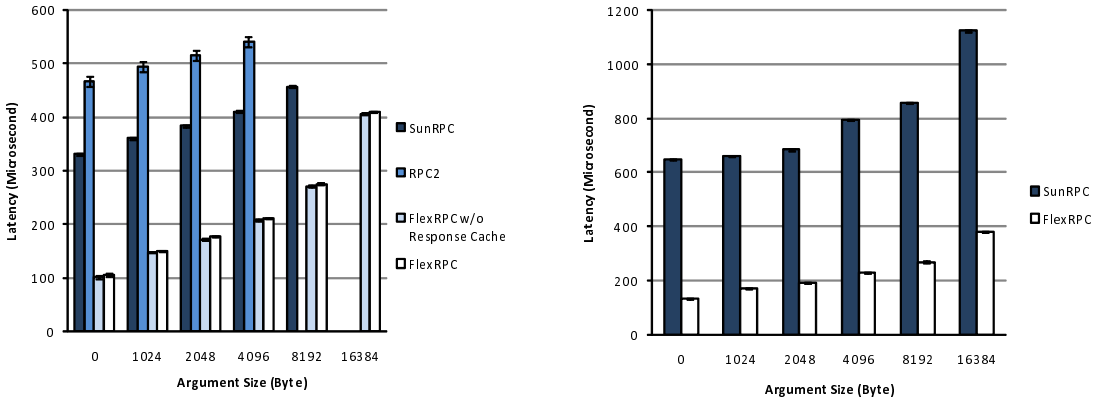
A. Methodology

The evaluation of FlexRPC has been performed on eight Linux machines. Each machine consists of 3GHz Pentium D processor, 2GB of main memory, and an on-board Broadcom NetXtreme BCM5721 Gigabit Ethernet network interface card. They are connected with a 3com Baseline 2824 Gigabit Ethernet switch. We turned on the HyperThreading option in the processor and the SMP version of Linux kernel 2.6.15-1 was used as operating system. A machine is dedicated to the caller and the rest of the machines are used as the callees.

We have compared the performance of FlexRPC with that of SunRPC in glibc 2.4-4 and RPC2 1.27-1. We have also built working prototype of a cluster file system called *Kadoop* on top of FlexRPC. We present the file read/write performance of Kadoop and demonstrate the flexibility of FlexRPC.

TABLE II
A COMPARISON BETWEEN SUNRPC AND FLEXRPC

Category	Purpose	SunRPC	FlexRPC
Types	Handle	`CLIENT`	`frpc_handle`
	Request Descriptor	`svc_req`	`frpc_request`
Interfaces	Get Connection	`CLIENT * clnt_create()`	`frpc_handle * frpc_createhandle()`
	Release Connection	`void clnt_destroy()`	`void frpc_destroyhandle()`
	Manage Connection	`bool_t clnt_control()`	`bool_t frpc_control()`
	Invoke RPC	`int clnt_call()`	`int frpc_call()`
	Register service	`void registerrpc()`	`int frpc_register_program()`
	Start callee service	`void svc_run()`	`void frpc_svc_run()`



(a) UDP

(b) TCP

Fig. 4. Single call latency

B. Single Call Latency

We analyze the average latency by measuring the total elapsed time to complete 100 RPCs with six different argument sizes varying from 0 to 16 KB over UDP and TCP. The target remote procedure does nothing returning zero-byte result to the caller immediately. Each experiment was repeated 10 times. The payload was chosen to be opaque data type and all authentication and encryption options were turned off to minimize the effect of XDR (eXternal Data Representation) and encryption layer. For UDP, the performance of FlexRPC without response cache has been also measured to identify the overhead of response cache.

Figure 4 summarizes the results. The standard deviations are displayed with error-bars. Note that SunRPC and RPC2 are incapable of handling argument sizes larger than 16 KB and 8 KB, respectively, on UDP. In addition, we are unable to show the results of RPC2 on TCP since RPC2 does not support TCP transport.

We can observe from Figure 4 that FlexRPC shows the shortest latency for all the tested cases. RPC2 suffers from the longest latency per call possibly due to the complex inner structure and protocols. Compared to SunRPC, FlexRPC improves the latency by up to 68% on UDP and by up to 79% for TCP. The use of portmapper and the lack of handle manager significantly increases the latency of SunRPC especially for TCP.

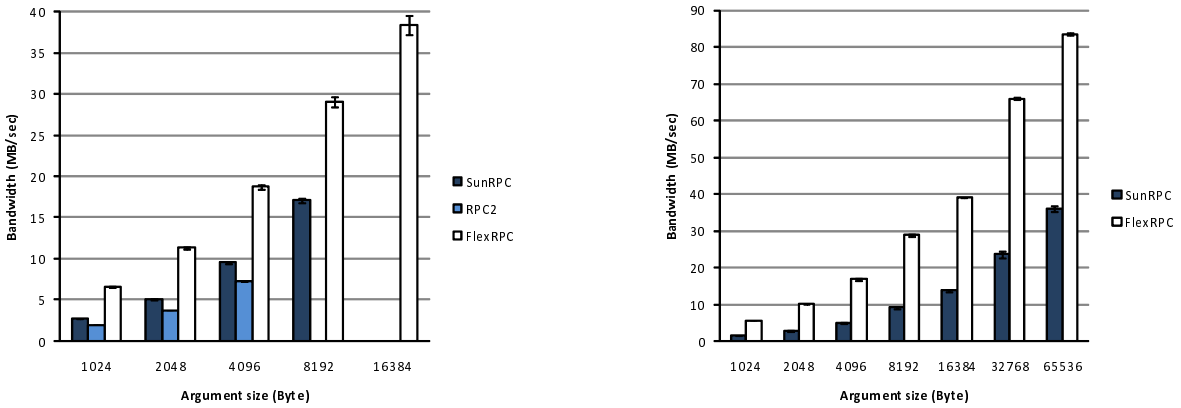
The response cache in FlexRPC places very little overhead on the latency. On average, only 5 microseconds is added to the latency due to the response cache regardless of the argument size. This amount of latency is negligible considering that the latency goes beyond 100 microseconds.

C. Single Call Bandwidth

The results obtained in Section V-B can be used to calculate the bandwidth of single calls. Figure 5 depicts the sum of the transmitted argument divided by the elapsed time. We do not count the zero-byte result. Figure 5 includes additional results of argument sizes larger than 16 KB for TCP.

It is obvious that the shorter latency results in the higher bandwidth. As a result, FlexRPC exhibits the highest bandwidth. Note that FlexRPC over UDP and FlexRPC over TCP show nearly the same bandwidth while it is not the case for SunRPC. This is because the connection overhead is eliminated in FlexRPC due to handle caching.

Figure 6 illustrates the aggregated bandwidth when the number of concurrent callers varies from one to six threads with 16 KB argument size. The aggregated bandwidth is defined as the sum of the bandwidth of each caller. We have measured the aggregated bandwidth for FlexRPC only because SunRPC cannot handle concurrent RPC requests and RPC2 does not satisfy client-side thread-safeness. As Figure 6 shows, FlexRPC utilizes full network bandwidth if the number of concurrent threads becomes larger than three.



(a) UDP

(b) TCP

Fig. 5. Single call bandwidth usage

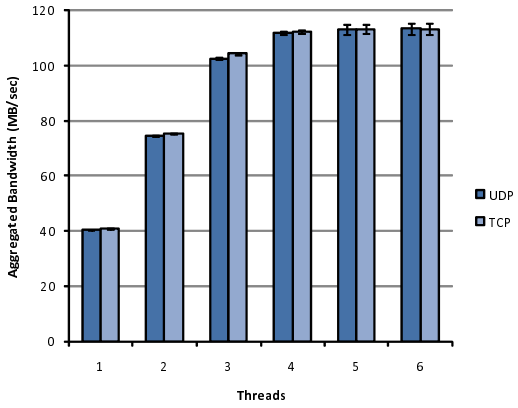


Fig. 6. Aggregated bandwidth in FlexRPC (16 KB argument size)

D. Multicasting Call Bandwidth

We have measured the effective bandwidth in order to quantify the performance of multicasting calls. The effective bandwidth is calculated by the sum of the argument size divided by the time to complete multicasting calls. We invoked 100 multicasting calls to three callees (i.e., multicasting degree of three) and repeated the same experiment ten times. For parallel multicasting, the actual bandwidth used by the caller is three times of the effective bandwidth since the same argument is transferred to three callees in parallel.

Figure 7 depicts the effective bandwidths of SunRPC, RPC2, and FlexRPC. SunRPC does not support any kind of multicasting call and the results are obtained by making three consecutive single calls. Recall that RPC2 does not support TCP and FlexRPC implements serial multicasting over TCP only.

From Figure 7, we can see that FlexRPC outperforms SunRPC and RPC2 significantly in both UDP and TCP. The effective bandwidth of parallel multicasting in FlexRPC over UDP has been improved by up to 79% compared to MultiRPC in RPC2. It is surprising that RPC2, whose MultiRPC facility is specialized for parallel multicasting, achieves only marginal benefit. Figure 7 shows that the design of FlexRPC based on multiple calling workers is very lightweight and effective in implementing multicasting calls.

In FlexRPC, parallel multicasting reveals better effective bandwidth for argument sizes smaller than 16 KB, while serial multicasting outperforms parallel multicasting for argument sizes larger than 16 KB. This is because the congestion on the network during parallel multicasting limits the overall performance for large argument sizes. The situation becomes worse as the number of concurrent callers or the multicasting degree increases.

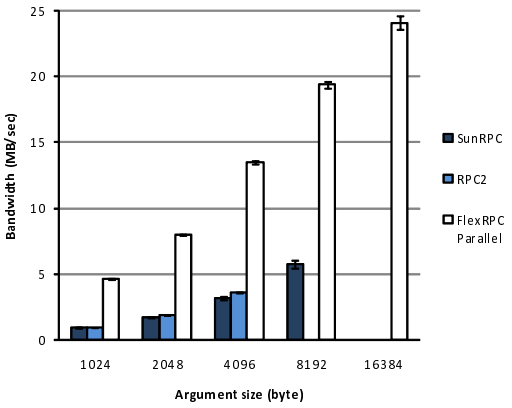
In Figure 8(a), we investigate changes in the aggregated effective bandwidth as the number of concurrent callers varies from one to ten in FlexRPC over TCP. The aggregated effective bandwidth represents the sum of the effective bandwidth shown by each caller. It is clear that even two concurrent callers nearly saturate the whole network for 32 KB argument size in parallel multicasting. The aggregated effective bandwidth is kept at about 37 MB/sec beyond the saturation point. On the other hand, serial multicasting shows the scalable aggregated effective bandwidth up to four concurrent callers. Serial multicasting achieves up to 94 MB/sec which is 2.6 times better than parallel multicasting.

Figure 8(b) exhibits changes in the effective bandwidth as the multicasting degree increases. We can notice that serial multicasting outperforms parallel multicasting if the multicasting degree is larger than one. Serial multicasting improves the effective bandwidth by up to 35%.

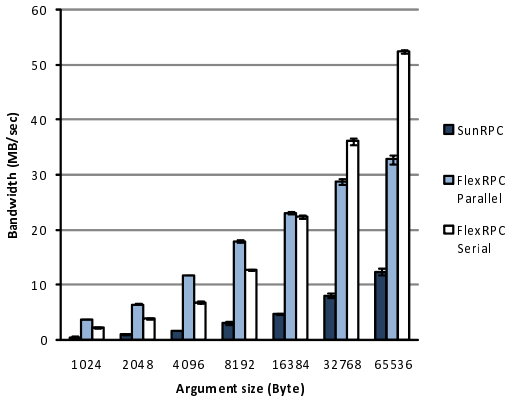
The results in Figure 7 and Figure 8 indicate that, during the development of cluster file systems, parallel multicasting is best suited to metadata operations with small argument sizes while serial multicasting is suitable for file replication operations which require large argument sizes.

E. Performance of Prototype Cluster File System

To show the effectiveness of FlexRPC, we have built working prototype of a cluster file system on top of FlexRPC, called

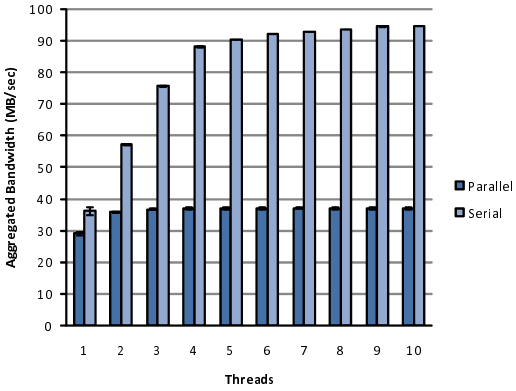


(a) UDP

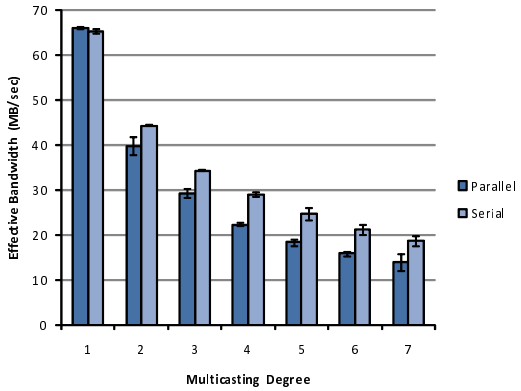


(b) TCP

Fig. 7. Effective bandwidth for multicasting calls



(a) Changes in the aggregated effective bandwidth (32 KB argument size with multicasting degree of three)



(b) Changes in the effective bandwidth (single caller with 32 KB argument size)

Fig. 8. Effective bandwidths of multicasting calls in FlexRPC over TCP

Kadoop. Kadoop is a port of HDFS [17] release 0.7.2 in C language and supports almost the same requirement specification of HDFS. Files are divided into 64 MB chunks and the chunks are replicated over arbitrary number of *datanodes*. Single *namenode* manages the metadata of the file system. Namenode is recovered from failure with checkpointing and write-ahead logging, and the chunks on failed datanode are migrated to other live datanodes.

The communication layer of Kadoop is based on FlexRPC over TCP. The replicas are distributed among datanodes using serial multicasting. The whole system of Kadoop has been built within a month by two developers. The total line count of Kadoop is 7,834 lines where the line count of the file system portion excluding the supplemental library is only 5,043 lines. The RPC-level support for connection management and pipelined data transfer reduced the development cost significantly.

We have measured the aggregated effective read and write bandwidths of Kadoop. The write bandwidth is measured by replicating one hundred 64 MB files to three datanodes. Each file is replicated by 64 serial multicasting calls, where each call sends out successive 1 MB of file contents. The read bandwidth is obtained by reading randomly chosen file 1600 times out of one hundred written files. Each time the target datanode is also selected randomly from three available replicas. A file is read by 64 single calls in a unit of 1 MB. One namenode and six datanodes are used for running Kadoop, and one client initiates read and write requests using Kadoop DFS library varying the number of concurrent threads from one to five.

Figure 9 shows the resulting performance of Kadoop. The read performance is scalable up to five concurrent threads achieving nearly up to 100 MB/sec. The write performance is, however, saturated to about 80 MB/sec mainly due to the replication across three datanodes. Kadoop demonstrates that FlexRPC is indeed a flexible, easy-to-use RPC facility for building high-performance cluster file systems.

VI. Conclusion

This paper presents the design issues and implementation details of FlexRPC. We have identified five crucial require-

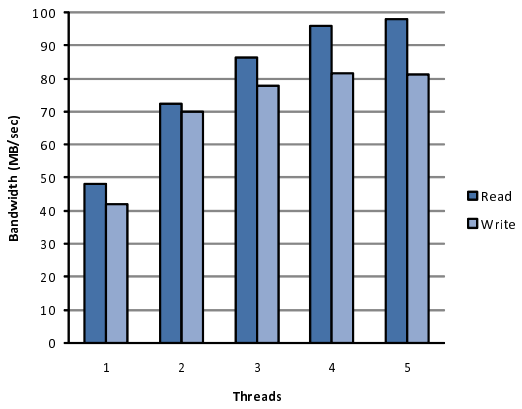


Fig. 9. Aggregated effective read and write bandwidths of Kadoop

ments that are considered necessary for implementing modern cluster file systems: client-side thread-safeness, support for multithreaded RPC servers, support for parallel and serial multicasting calls, support for at-most-once semantics, and support for various transports including UDP and TCP. FlexRPC is designed to meet such requirements.

FlexRPC ensures client-side thread-safeness and fully supports multithreaded RPC servers. Parallel and serial multicasting mechanisms allow for implementing sophisticated replication in modern cluster file systems. The remote procedure can be invoked using both UDP and TCP transports with at-most-once semantics. The concurrent call requests are handled by a set of worker threads on the client and server side where the number of workers varies dynamically according to the request rate. In addition, the semantics and the specification of remote procedures are designed to be as close as possible to SunRPC.

Our experimental results show that FlexRPC improves both latency and bandwidth significantly. The latency is reduced by up to 68% over UDP and by up to 79% over TCP compared to SunRPC. The bandwidth of parallel multicasting has been improved by up to 79% compared to MultiRPC in RPC2. We have also demonstrated the performance and the flexibility provided by FlexRPC by building working prototype of cluster file system called Kadoop on top of FlexRPC.

Using FlexRPC, we are currently developing a large-scale high-performance cluster file system which runs on hundreds of commodity hardware providing hundreds of terabytes of aggregated storage capacity. Further optimization on FlexRPC and the communication support between heterogeneous computer systems are left as future work.

ACKNOWLEDGEMENT

This work was supported by NHN Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsor.

This work was also supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government(MOST) (R01-2007-000-11832-0)

REFERENCES

[1] J. E. White, *A High-Level Framework for Network-Based Resource Sharing*. RFC707, The Internet Engineering Task Force, 1976.
[2] B. J. Nelson, "Remote Procedure Call," *Tech. Rep. CSL-81-9, Xerox Palo Alto Research Center*, 1981.
[3] X. Corp., "Courier: The Remote Procedure Call Protocol," *Xerox System Integration Standard 038112*, 1981.
[4] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. on Computer Systems*, vol. 2, no. 1, pp. 35–59, 1981.
[5] S. M. Inc., *RPC: Remote Procedure Call Protocol Specification*. RFC1057, The Internet Engineering Task Force, 1988.
[6] R. Srinivasan, *RPC: Remote Procedure Call Protocol Specification Version 2*. RFC1831, The Internet Engineering Task Force, 1995.
[7] B. N. Bershad, T. E. Anderson, and E. D. Lazowska, "Lightweight Remote Procedure Call," *ACM Trans. on Computer Systems*, vol. 8, no. 1, 1990.
[8] M. Satyanarayanan, *RPC2 User Guide and Reference Manual*, Carnegie Mellon University, 1991.
[9] M. Satyanarayanan and E. H. Siegel, "Parallel Communication in a Large Distributed Environment," *IEEE Trans. on Computers*, vol. 39, no. 3, March 1990.
[10] A. L. Ananda, B. H. Tay, and E. K. Koh, "ASTRA - An Asynchronous Remote Procedure Call Facility," in *Proc. 11th Int'l. Conf. on Distributed Computing Systems*, 1991.
[11] T. H. Dineen, P. J. Leach, N. W. Mishkin, J. N. Pato, and G. L. Wyant, "The Network Computing Architecture and System: an Environment for Developing Distributed Applications," in *Proc. 33rd IEEE Computer Society Int'l Conference (Compcon)*, 1988, pp. 296–299.
[12] R. Sandberg, "Design and Implementation of the Sun Network Filesystem," *Proc. USENIX 1985 Summer Conference*, June 1985.
[13] J. Howard, "An Overview of the Andrew File System," *Tech. Rep. CMU-ITC-88-062*, 1988.
[14] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Trans. on Computers*, vol. 39, no. 4, April 1990.
[15] Cluster File Systems, Inc., *Lustre: A Scalable, High-Performance File System*, http://www.clusterfs.com.
[16] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proc. 19th ACM Symp. on Operating Systems Principles*, 2003.
[17] The Hadoop Opensource Project. [Online]. Available: http://lucene.apache.org/hadoop/
[18] The GNU Operating System. [Online]. Available: http://www.gnu.org
[19] The OpenSolaris Project. [Online]. Available: http://src.opensolaris.org
[20] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang, "The Case for a Single-Chip Multiprocessor," in *Proc. 7th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 2–11.
[21] B. H. Tay and A. L. Ananda, "A Survey of Remote Procedure Calls," *Operating Systems Review*, vol. 23, no. 3, pp. 68–79, July 1990.
[22] A. L. Ananda and B. H. Tay, "A Survey of Asynchronous Remote Procedure Calls," *Operating Systems Review*, vol. 26, no. 2, pp. 92–107, Apr. 1992.
[23] D. Mazieres, M. Kaminsky, M. F. Kaashoek, and E. Witchel, "Separating Key Management from File System Security," in *Proc. 17th ACM Symp. on Operating Systems Principles*, 1999.
[24] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications," *IEEE/ACM Trans. on Networking*, vol. 11, no. 1, pp. 33–46, 2003.
[25] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Low-bandwidth Network File System," in *Proc. 18th ACM Symp. on Operating Systems Principles*, 2001.
[26] R. von Behren, J. Condit, and E. Brewer, "Why Events Are a Bad Idea (for High-Concurrency Servers)," in *Proc. 9th Workshop on Hot Topics in Operating Systems*, 2003, pp. 19–24.
[27] M. Herlihy, "A Quorum-consensus Replication Method for Abstract Data Types," *ACM Trans. on Computer Systems*, vol. 4, no. 1, pp. 32–53, 1986.