Evolving RPC for Active Storage

Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin–Madison
{muthian, dusseau, remzi}@cs.wisc.edu

ABSTRACT

We introduce Scriptable RPC (SRPC), an RPC-based framework that enables distributed system services to take advantage of active components. Technology trends point to a world where each component in a system (whether disk, network interface, or memory) has substantial computational capabilities; however, traditional methods of building distributed services are not designed to take advantage of these new architectures, mandating wholesale change of the software base to exploit more powerful hardware. In contrast, SRPC provides a direct and simple migration path for traditional services into the active environment.

We demonstrate the power and flexibility of the SRPC framework through a series of case studies, with a focus on active storage servers. Specifically, we find three advantages to our approach. First, SRPC improves the performance of distributed file servers, reducing latency by combining the execution of operations at the file server. Second, SRPC enables the ready addition of new functionality; for example, more powerful cache consistency models can be realized on top of a server that exports a simple NFS-like interface. Third, SRPC simplifies the construction of distributed services; operations that are difficult to coordinate across client and server can now be co-executed at the server, thus avoiding costly agreement and crash-recovery protocols.

1. INTRODUCTION

Remote Procedure Call (RPC) has long been the standard for implementing distributed services [6]. RPC is simple to use, as it extends the well-known paradigm of procedure call to a client/server setting, and yet powerful enough to serve as the substrate for many distributed services (*e.g.*, Sun's NFS). However, since the development of RPC, little has changed in the functionality it offers; modernized versions of RPC (such as Java RMI [33]) do not significantly alter the basic RPC paradigm, but simply provide the same features within a different language or run-time context.

While RPC is stagnating, the architecture of distributed systems is changing rapidly. In particular, technology trends point to a world where "active" components are commonplace. Intelligence in the form of additional processing capabilities is or will soon be found in active disks [1, 18, 37], smart network interfaces [7, 16], and even intelligent memories [29, 36].

Of particular importance is active storage, which has the potential to greatly improve distributed storage services, in terms of performance, functionality, and even simplicity of design. Although early research on active storage has thoroughly demonstrated its potential performance benefits [1, 2, 18, 37], previous systems require entirely new programming environments, and some target only a limited class of applications (*e.g.*, parallel applications and database primitives). Existing distributed file services have no clear migration path into the coming active world.

In this paper, we present the design, implementation, and evaluation of Scriptable RPC (SRPC), an extensible framework for developing active distributed services. The key difference between SRPC and traditional RPC is that clients can send scripts to the server to implement new server-side functionality. SRPC is designed to assist developers in evolving traditional RPC-based services into first-class active services; therefore, SRPC shares many features with RPC, and automates as much of the service development process as possible to increase ease-of-use. For example, SRPC automatically embeds the script interpreter within the server, providing a safe execution environment for client-generated scripts.

We demonstrate three primary benefits of SRPC through a number of case studies within a distributed file system called ScFS. First, ScFS improves performance by combining various operations at the server, thus avoiding the costs of extra round-trip latencies. Second, ScFS allows the ready addition of new functionality, evolving a base file system protocol into an enhanced virtual protocol. Specifically, scripts are applied to implement AFS and Sprite cache consistency on top of a base protocol that provides only simple NFS-like consistency. Finally, ScFS provides a simple way to implement functionality that would be complex to execute reliably across both client and server. In ScFS, directory update operations are performed in a script directly at the server, and thus obviate the need for multi-client coordination.

When introducing scripting into a file server, a number of issues arise. One concern is safety [5]; client kernels, though partially trusted, should not be able to induce the server to crash (whether accidentally or with malicious intent) or monopolize its resources. Performance is also an issue, as script interpretation overhead could negate many of the advantages of SRPC. Finally, ease-of-use is an important criterion, as the language should enable the development of short but powerful scripts.

In our prototype implementation of SRPC, we use Tcl for our base scripting language [35] – many similar languages exist, but we feel that Tcl is an interesting choice for three reasons. First, Tcl addresses safety concerns by providing a limited execution environment, with fine-grained control over which operations a script can execute (*e.g.*, even `while` loops can be disabled). Second, although previous studies indicate that the Tcl interpreter is many thousand times slower than C [38, 40], more recent versions of the Tcl interpreter are quite sophisticated, and use efficient internal representations to boost performance. Thus, the time is ripe for reexamining the strengths and weaknesses of Tcl performance. Third, Tcl is a fairly high-level language; ease-of-use should be one of its main advantages.

Overall, we find that SRPC is an effective framework for extending the ScFS distributed file service. Performance and functionality enhancements can readily be designed and deployed into the file system framework. In some of our case studies, we find that using scripts can reduce latency by a factor of two and strengthen the consistency semantics of the file system. However, although server functionality is readily embellished within the SRPC framework, we find that a similar extension methodology for client-side kernels would also be useful. In our evaluation of Tcl as the language for extensibility, we find that it is indeed powerful while remaining easy to use – even the most complex extension requires only a few tens of lines of code. We also find that the performance of Tcl has improved enough to warrant consideration as a language used for extension, especially given its numerous safety features. However, some of our more sophisticated extensions cannot afford even the slightest of overheads, and therefore, a specialized domain-specific language for active storage may be worth investigating.

The rest of this paper is structured as follows. We present an overview of our SRPC system in Section 2, and describe our experimental environment, including details on ScFS and our empirical methodology, in Section 3. In Section 4, Section 5, and Section 6, we illustrate the manner in which SRPC can be used to improve the performance, functionality, and simplicity of active file services, respectively. In Section 7, we perform an in-depth analysis of our extension scripts and Tcl. We then cover related work in Section 8 and conclude in Section 9.

2. OVERVIEW OF SRPC

SRPC is a new framework for building active, extensible distributed services. SRPC extends the widely used and well understood paradigm of RPC with a flexible scripting capability that clients can harness to execute customized extensions at the server.

SRPC is designed to meet three goals that are important for this environment. The first goal is to provide a smooth migration path for existing distributed services from a traditional RPC-based server to an active server. This goal is accomplished by making the process of developing an SRPC-based service similar to that of developing a service within the traditional RPC paradigm, automating as many of the steps as possible. The second goal is to ensure script execution overhead is not severe as compared to a traditional RPC call. To improve performance, SRPC caches scripts, supports concurrency, and provides efficient implementations of "performance-sensitive" commands in the SRPC standard library. The third goal is to provide a safe environment for script execution; our choice of Tcl as a scripting language enables us to meet this goal.

2.1 Migration Path

The first goal of SRPC is to enable developers to easily move distributed services using traditional RPC-based servers to more powerful, active servers. Thus, SRPC is engineered to be backwardly compatible with RPC, allowing unmodified clients (that do not send scripts to the server) to co-exist with clients that have been altered to use this new functionality. Further, SRPC automatically generates the necessary code to interface between the RPC server and the scripting language interpreter.

In traditional RPC, the developer of a distributed service defines the interface that the server exports and specifies this interface in an Interface Definition Language (IDL). This interface definition is parsed by an IDL compiler to generate source code for the client and server stubs that call into the RPC library. The developer only needs to implement the procedures specified in the interface, which can then be compiled together with the automatically generated stubs to produce the distributed service.

Developing an active distributed service in SRPC involves a similar set of steps. The basic interface exported by the server is still specified with an IDL, but the functionality of the IDL compiler is augmented in three ways. First, a new RPC procedure is added to the interface called `ScriptExec`, which takes as parameters a script and a data buffer and returns a data buffer as a result; the implementation of `ScriptExec` is generated automatically as well. Second, the IDL compiler automatically generates Tcl wrappers for the interface procedures, so that client scripts can directly call these routines; to further simplify SRPC programming, the interpreter at the server provides commands to extract specific parameters from the character buffer used to pass arguments. Third, the compiler generates code to initialize a set of Tcl interpreters and registers the interface procedures with each interpreter, increasing ease-of-use and reducing the burden on programmers.

To further assist programmers in the development of scripts, there are two additional pieces of functionality that are supported within the SRPC infrastructure. First, in some scripts, it is useful to maintain *state* that can be later accessed from other scripts and by other clients; thus one script can store state under a given name and another script can access this state by specifying the name. State variables must be uniquely named such that different scripts do not inadvertently conflict on the same name. In the current SRPC implementation, the responsibility of namespace management rests with the clients, who follow certain pre-defined conventions so that other clients can easily find and access state variables. The second piece of useful functionality is the ability for the server to invoke functionality on the client, *i.e.*, to perform a callback. SRPC supports this by allowing the client to run an RPC server as well.

2.2 High Performance

In order for SRPC to be an appealing platform for distributed systems, its performance must be comparable to that of a traditional RPC service. However, script interpretation imposes an overhead every time a script is executed, potentially negating some of the benefits of executing operations at the server. SRPC uses three simple techniques to improve performance: caching, concurrency, and a standard library of primitives.

The performance of initial versions of Tcl has been shown to have significant overhead [38]; however, much of this overhead occurred because every line of a script was re-interpreted upon execution. For efficient repeated execution of the same code, more recent versions of Tcl translate interpreted code into a more efficient internal form. To leverage this behavior, SRPC identifies scripts that have been executed previously and reuses the cached procedures for subsequent invocations (though perhaps with different arguments). Thus, instead of sending a script with each invocation, clients register scripts with the server, and get back a script ID; the ID can be passed to the server along with script arguments upon subsequent invocations. This caching mechanism has the further advantage of reducing the size of messages between the client and server, thus improving observed network latency and reducing the amount of network traffic. Our case studies in the next section quantify the performance benefits of SRPC script caching.

To further improve performance, the SRPC framework executes multiple Tcl interpreters concurrently. With multiple interpreters, scripts execute in parallel at the server, greatly improving system throughput. However, concurrent execution complicates the development of scripts, as we discuss when describing safety issues.

The third performance optimization within SRPC is to implement "performance-sensitive" operations inside the SRPC standard library, instead of directly within each script. Given that Tcl supports the calling of functions written in the C language, providing this functionality is straightforward. The issue is to identify those operations that should be included in the standard library, which in general are those operations that are both popular across scripts and costly to implement in Tcl. The current implementation of SRPC includes a standard library with routines to manipulate data buffers, search through buffers for a string, send messages, create and manage lists, access shared script state, and acquire and release locks. Of particular importance are routines that enable scripts to avoid touching data inside of Tcl; by ensuring that data buffers are solely manipulated within the C substructure [46], we avoid a primary source of additional overhead. Given that implementing server routines directly in C instead of Tcl bypasses many of the security issues of the scripting language, we anticipate that only trusted administrators will be able to add new primitive routines.

2.3 Safety

The third goal of the SRPC design is to ensure that scripts sent to the server do not easily corrupt server state or consume undue resources. The same problem arises in extensible operating systems research [5, 15, 39]; however, in SRPC, we do not allow arbitrary user applications to insert code into the server; instead, only kernel clients can do so, and thus, the trust boundary is more relaxed. Given this boundary, we still believe that it is advantageous to enforce a limited execution context for scripts.

Fortunately, Tcl provides the needed functionality required to limit script actions. With the SafeTcl extensions, the server can control which functions a script can call. Because control constructs are implemented as Tcl procedures, we can prevent the execution of `while` and `for` loops, and thus ensure that scripts terminate in a finite amount of time. For scripts that require iteration, we provide a simple yet safe callback interface, which allows a procedure to be executed some finite number of times.

One complication to the safety model is that scripts manipulate buffer pointers directly. To ensure that no invalid memory references are generated, SRPC implements various run-time checks inside all wrapper and standard-library routines, wherever pointers are passed as arguments. Run-time type checking within the automatically generated wrappers ensures that argument addresses correspond to the correct type, thereby preventing illegal dereferencing of arbitrary memory addresses. This additional level of safety comes at a cost, but it is one we believe to be worthwhile.

Another complication arises due to concurrency. Concurrent execution mandates the use of locks, to allow scripts that access shared state to execute correctly. However, locking introduces new problems within the realm of safe extensibility. For example, a client could misbehave and never release a lock, thus negatively impacting other well-behaved clients [39]. To prevent this problem from occurring, SRPC automatically releases any locks that were acquired and not released by an executing script upon script completion. Although this safeguard restricts the scope of locks, we have not found it to be a burdensome limitation.[1]

[1]Access control to locks would also be useful, prohibiting misbehaving clients from acquiring locks to which they are not privy. However, we have not yet implemented such functionality.

```
proc SpriteCB {ARG RES ip state name lock} {
  # global variable 'c'
  global c

  # load the 'cacheable' flag
  set cache [srpc_load_state $state]

  if {$cache} {
    # add client to callback list
    set list [srpc_load_state $name]
    if {$list == -1} {
      set list [srpc_makelist]
      srpc_install_state $list $name
    }
    srpc_add_to_list $list $ip
  }
  # get args/invoke read
  set rdarg [scfs_make_rdarg $ARG $c(rdOff)]
  set rdres [scfs_read $rdarg]

  # extract components of result
  set cnt  [scfs_rdres_getlen $rdres]
  set data [scfs_rdres_getdata $rdres]

  # pack state/results into result
  srpc_putInt $RES $c(cacheOff) $cache
  srpc_putInt $RES $c(cntOff) $cnt
  srpc_memcpy $RES $c(dataOff) $data $cnt

  return [expr $cnt+$c(cacheSz)+$c(cntSize)]
}
```

Figure 1: Example SRPC Script. *This script performs a read of a file in the presence of write-sharing, and is one of three scripts used to enforce "Spritely" cache consistency. As one can see, augmenting a stateless protocol with stateful functionality involves only a handful of Tcl commands.*

2.4 Example

In Figure 1, we present an example of an SRPC script. This script is part of the Sprite consistency implementation, and is executed by a client when it reads a file marked uncacheable – the example is explored in more detail as a case study in Section 5.2.

Given that the RPC interface at the server exports a `read` procedure, the IDL compiler automatically generates the corresponding Tcl wrapper `scfs_read`. Commands to extract individual fields (*e.g.*, `scfs_rdres_getdata`) and to make arguments of a given type from a character buffer (*e.g.*, `scfs_make_rdarg`) are also automatically generated, and type checking is performed within to ensure that the script cannot access invalid memory addresses. Examples of standard library routines include `srpc_loadstate` and `srpc_putInt`.

The script begins by loading the cacheable flag corresponding to the object of interest (*i.e.*, a file). If set, the IP address of the client is added to the object's callback list. The script then creates a valid read argument from the data passed into the script (accessed through the `ARG` variable) and invokes `read` with the argument. The results of the read, together with the cacheable flag, form the result of this script, which is packed into the result variable `RES` and returned to the caller.

3. EVALUATION ENVIRONMENT

We evaluate the SRPC framework in the context of a distributed file system known as ScFS. In this section, we begin by presenting a general overview of ScFS. We then describe the details of our experimental platform. Finally, we describe a mechanism to systematically increase network delay in order to understand the performance of SRPC in different environments.

3.1 The Scripted File System (ScFS)

Distributed file systems built over network-attached disks have the potential of delivering high bandwidth without the additional cost of servers [17, 18]. Thus, to evaluate the benefits of SRPC, we introduce ScFS, a scripted distributed file system for network-attached storage. The base version of ScFS supports multiple clients and a single disk, which acts as a repository for all file system data. ScFS exports a hierarchical namespace to applications, performs basic caching, allows multiple outstanding requests to tolerate latency, and implements an NFS-like weak cache consistency scheme. Throughout this paper, we demonstrate how the scripting capability provided by SRPC can enhance the performance and functionality of ScFS.

In our prototype, a network-attached disk can be configured to export one of two interfaces to clients. The first and most commonly used is an object-based interface similar to that proposed by Gibson *et al.* [19]. The exported object namespace is non-hierarchical and does not distinguish between directories and normal files. Read and write requests are arbitrarily fine-grained, specifying an object identifier, offset, and length. The second is a block-based interface, which allows us to explore the capabilities of SRPC within a more restricted legacy environment. In this mode, disk reads and writes are all 4 KB in size.

Clients communicate with the disk through a protocol that is similar to NFS, with the disk server not maintaining any state on behalf of clients. A slight difference is that NFS calls for directory operations do not exist, since the clients perform these operations themselves. Each client also runs an in-kernel RPC server to receive messages sent by scripts executing at the disk.

3.2 Experimental Platform

Our system currently runs on a testbed of Intel-based machines, each running the Linux 2.2.19 operating system. The ScFS client-side code is developed in the kernel within the standard Linux vnode interface. Each PC contains a 550 MHz Pentium III, 1 GB of memory, and a 9.1 GB IBM Ultrastar 9LZX disk. The machines are connected together via 100 Mbit/s Ethernet and Gigabit Ethernet, but our experiments primarily use 100 Mbit/s Ethernet.

In our environment, a network-attached disk is emulated by a PC. Though the exact capabilities of a PC may not match that of a network-attached disk (*e.g.*, the processor in a network-attached disk may be engineered for low power consumption and not absolute performance), we believe this is a reasonable approximation of a network-attached storage unit. Further, some believe that disks will soon have high-end processing capabilities, rivaling the CPU power available in a commodity PC [21].

3.3 Network Emulation Methodology

The potential performance benefits of active disk servers may depend on the exact characteristics of the environment. If perceived network latency is quite low, there is a smaller performance benefit to combining operations in a script. In contrast, in a high-latency scenario, whether across the wide-area or over a dial-up link from home, the performance benefits will be much larger [31].

To study the impact of the network on the performance of ScFS we have implemented a framework for increasing network latency by a controlled amount. Our approach is similar to that introduced by Martin *et al.* in their study of the effects of latency, overhead, and bandwidth on parallel applications [30]. Specifically, to delay a packet, the server queues the packet internally and records the time at which the packet was received. A thread within the server removes messages from the queue when the designated delay has passed and services them. Experiments (not shown here) validate that our delay mechanism behaves as desired.

4. PERFORMANCE ENHANCEMENTS

Often, to achieve a single logical task in a network file system, a series of dependent interactions between the client and server are required. These dependencies impose synchrony on the client-server interactions, since one operation cannot be initiated before the previous one has returned. Thus, clients incur multiple network round-trips to accomplish a single logical task, which can result in significant performance loss, especially in high-latency networks or under severe server load.

ScFS uses the SRPC framework to group dependent operations into a script that is then executed at the disk in a single network round-trip. We illustrate the efficacy of our scripting infrastructure with three case studies. In the first study, we combine a client lookup operation of dependent `getattr` and `read` calls into a single script. In the second, we merge an NFS-like consistency check with the possible read of a modified page, similar to an HTTP get-if-modified-since request [4]. In the third, we avoid a read-modify-write cycle by sending a "partial write" script to the server.

4.1 Motivation

Before we discuss our three case studies, we first examine the effects of server load on perceived client latency. Network latency is a concern in the wide-area and over dial-up connections, where round-trip times of 10s of milliseconds are not uncommon [31]; however, even in a tightly-coupled cluster with a high-performance communication subsystem [44, 45], latency can still be a considerable factor due to server load. To illustrate the effects of server load on perceived round-trip time, we perform a simple experiment.

In the experiment, we measure the round-trip times observed by a single client communicating with a server using small (128 byte) messages. We vary the number of competing clients, each of which continually sends a stream of large requests (60 KB) to the server. With no competing processes, the perceived average round-trip time is quite reasonable, roughly 150 μs over our Gigabit Ethernet network. However, as traffic-inducing competitors are added, the perceived round-trip time for the client increases dramatically, to over 5 ms with four competing processes. Note that this effect is strictly due to queuing at the network interface; the server does not touch the data and its CPU is under-utilized.

The problem illustrated here, although somewhat obvious upon inspection, is often overlooked in the design of distributed systems for clusters. [2] The fact that messages can be transmitted quickly in an idle system does not avoid a *convoy effect*, where small messages queue behind large ones at the network interface, and thus do not promptly reach their destination. If response time is one's performance metric (and not throughput), server load can easily transform a low-latency network into something that is perceived much

[2]For example, in [22], the authors discuss assumptions made in their implementation of distributed cluster-based service: "[A cluster's] low-latency SAN (10-100 μs latency instead of 10-100 ms for the wide-area Internet) means that two-phase commits are not prohibitively expensive." Others have made similar assumptions in the design of cluster-based systems (*e.g.*, [10], page 5, Section 3).

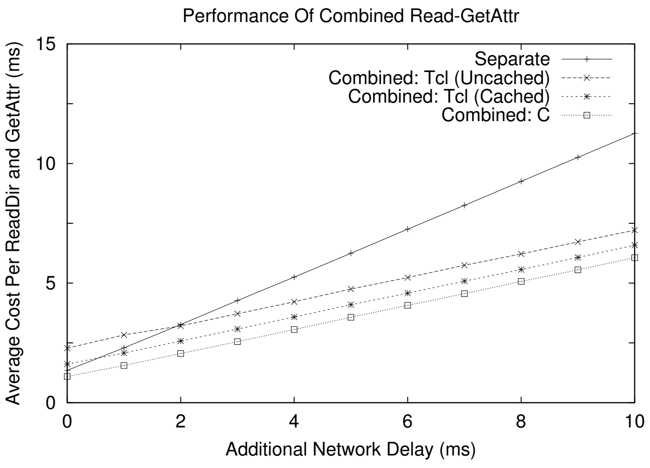


Figure 2: Combining Operations: `read` **and** `getattr`**.** *The figure shows the performance of combining* `read` *and* `getattr` *operations into a single script. The figure plots the average cost of performing a directory read with a subsequent attribute request, and we increase the perceived round-trip latency of the request along the x-axis, from 0 ms to 10 ms. The "Separate" line shows performance when two network round-trips are required, and the other three lines in the graph show the performance of combining those operations at the server. The "Combined: Tcl (Uncached)" line shows Tcl performance when the procedure is installed upon every request, and the "Combined: Tcl (Cached)" line shows performance when the script has been pre-installed. Finally, the "Combined: C" line gives a baseline for performance for a C implementation of the script. Each data point represents the average of 30 trials, and the variance (not shown) is low.*

differently by clients during operation under load. Thus, in the experiments in this section, we evaluate the performance of ScFS as a function of network latency.

4.2 Read-GetAttr Optimization

In our first case study, we illustrate the performance benefits of combining a directory read with getting the attributes of a particular file within that directory. Given the disk object model used as the basis of ScFS, a file lookup at the client involves reading each directory in the path followed by a search for the given path component. In the worst case, when no path components are in the local cache, two round-trip network operations are required for each component of the path: the first to read the directory page of the parent and the next to get the attributes of the child object to initialize the VFS inode.

In ScFS, the SRPC infrastructure at the disk enables the client to send a script that reads the directory page, searches for the specified filename to get the object ID, and then performs a `getattr` on the object. The script then returns the results of both the `read` and the `getattr` calls. Thus, the client now completes the lookup operation for each path component in one network round-trip instead of two. Although a client file system could be designed that performs the lookups for all components of the pathname in a single script (and thus avoids more network round-trips), this approach does not integrate easily into the Linux in-kernel framework.

To evaluate the trade-offs of using scripts for a `read-getattr` operation as a function of network latency, we compare the perceived latency of four different implementations in Figure 2. The first implementation, labeled "Separate" in the figure, is the standard implementation that synchronously issues two separate requests using RPC. The second, labeled "Combined: Tcl (Uncached)", uses a Tcl script to combine the two operations, but the script is installed in the interpreter at every invocation. The third, labeled "Combined: Tcl (Cached)", considers the case of a pre-installed script. The fourth implementation, labeled "Combined: C", combines the two operations but uses the C language; this version serves as a baseline for ideal combined performance and can be utilized to understand the overhead of scripting.

From the figure, we draw three conclusions. First, with a "fast" interpreted language (*i.e.*, one that approaches C speeds), combining `read` and `getattr` operations into a single script leads to lower overhead than the base implementation, regardless of the amount of network latency. Second, caching of Tcl procedures greatly reduces the cost per operation, removing almost half a millisecond of overhead per operation. Finally, at higher latencies, combining operations, even within a "slow" interpreter (*i.e.*, uncached Tcl procedures), is strictly better than the standard implementation. Note that these latencies, while on the higher end in a tightly-coupled cluster environment, are still low when considering a dial-up link [31], where latencies in the 10s of milliseconds would not be uncommon.

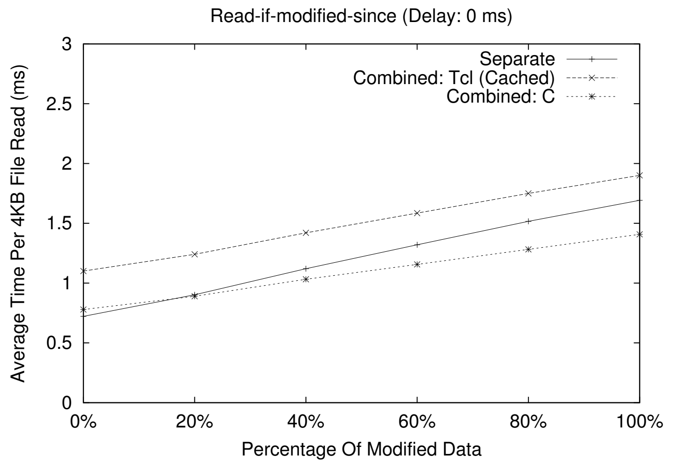
To demonstrate the utility of the `read-getattr` optimization within an more realistic benchmark, we utilize the PostMark benchmark [25]. PostMark is a file system benchmark constructed to mimic the workload of a typical mail server. It consists of a create phase followed by a transaction phase, in which files are randomly created, deleted, read and appended. To generate a more realistic workload, we make two changes to the PostMark benchmark. First, we separate the create phase from the transaction phase and present the results only for the transaction phase. Left unmodified, the creation phase warms the cache and thus artificially speeds up the transaction phase. The second change we make results in a more realistic layout of directories, instead of the default behavior of PostMark which creates all directories at the same level in the directory hierarchy.

Measurements reveal that the `read-getattr` optimization removes 456 round-trip messages by combining `read` and `getattr` pairs into combined `read-getattr` script calls. During the course of the benchmark, 5736 total messages are sent; thus, this simple optimization reduces the number of round-trip latencies incurred by the client by 8%.

4.3 Read-If-Modified-Since Optimization

In this next case study, we illustrate how the scripting interface can make NFS-style consistency checks more efficient. NFS provides a primitive form of consistency by periodically validating the cached copy on the client with the file server. If the time since the last validation is above a threshold, the cached copy is considered suspect; on a `read` request to a suspect copy, the client sends a `getattr` request to the server to check if the file has changed since it was cached. If the file has changed, the client invalidates its cached copy and fetches the pages from the server. These two operations are dependent since the client should not issue the `read` before knowing that the cached copy is stale, since it wastes network bandwidth to re-fetch valid pages.

In the SRPC framework, these two operations are merged in a single script that does a `getattr`, checks if the modification time of the file is higher than the specified modification time of the cached copy, and if so, sends the desired page of the file along with the attributes; we refer to this combined functionality as the "read-if-modified-since" operation. Again, with SRPC support, the



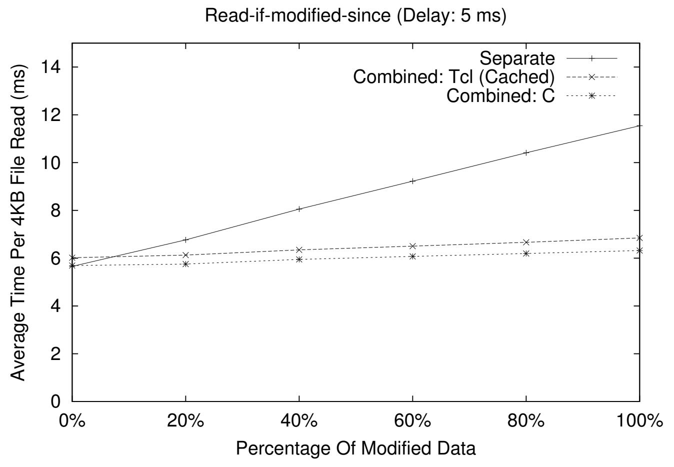


Figure 3: NFS Read-If-Modified-Since. *The graphs plot the performance of* `getattr` *followed by a conditional* `read` *of the data, if the data has changed since the last check. In the graph on top, we consider no additional network latency; in the graph on the bottom, we add a fixed network latency of 5 ms. Across the x-axis we increase the percentage of files that have been changed since the last check. Each data point is the average of 30 trials.*

file system performs the same logical task but incurs fewer synchronous network round-trip delays.

Figure 3 shows the performance of read-if-modified-since compared to the standard `getattr` followed by a conditional read. Along the x-axis of the graphs, we vary the percentage of data that has been modified since the last read and thus must be re-fetched from the server. In the graph on top, we plot performance assuming no additional network delay; in the graph on the bottom, we assume an additional network delay of 5 ms. We compare the performance of the standard implementation, a cached Tcl script, and the combined functionality implemented in C.

From the figure, we draw two conclusions. First, in the low-latency environment illustrated in the graph on top, the benefits of a combination script are small and only realized with a high-performance interpreter – the cached Tcl implementation adds too much overhead. Second, even in the higher-latency environment illustrated in the graph on the bottom, a relatively large fraction of files need to have changed for the Tcl implementation to perform better than the standard implementation (greater than 10% of the files). Thus, for optimizations that are only able to combine operations in limited circumstances, the overhead of interpretation is the dominant factor. Unless a higher performance interpreter is available, it is probably not worth implementing these kinds of features.

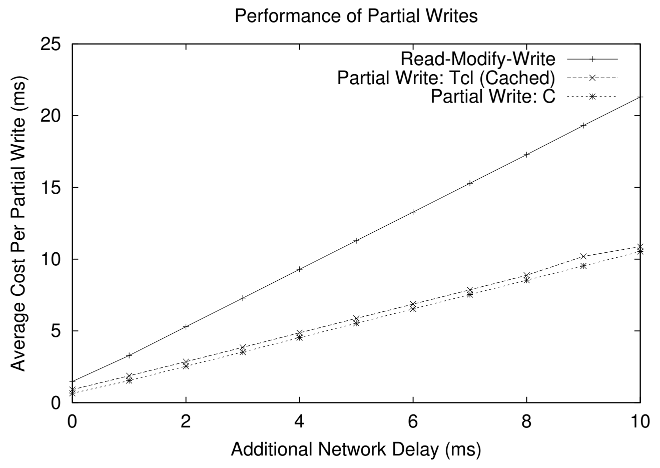


Figure 4: Partial Writes. *The figure shows the performance of performing a synchronous partial write in a script, instead of reading the data from the server, modifying it, and then writing the data out to the server. The figure plots the average cost of performing a synchronous 100-byte write, increasing the perceived latency of the request along the x-axis, from 0 ms to 10 ms. The "Read-Modify-Write" line shows performance in the standard read-modify-write case, and the other two lines show the performance of the scripted approach. The "Partial Write: Tcl (Cached)" line shows Tcl performance when the script has been pre-installed, and the "Partial Write: C" line gives a baseline for performance for a C implementation of the script. Each point is the average of 30 trials.*

4.4 Partial-Write Optimization

Writes that are less than the size of a page can be problematic in traditional systems, as each "partial-page write" translates into a read-modify-write cycle – a read of the relevant page into the cache, a modification to the affected portion of the page, and a write to complete the update. The read-modify-write cycle occurs in block-based systems, which export read and write operations on a page granularity.

Within the SRPC framework, we can avoid the read-modify-write cycle and instead send the data to be written and a script that performs the partial write on the server itself, thus removing the synchronous page-sized read from the critical path [2]. Figure 4 displays the performance of the server-side execution of the partial-write as compared to the standard read-modify-write cycle.

As one can see from the figure, avoiding the synchronous page read has great benefit, even without any additional network delay. Avoiding the excessive data movement of the read-modify-write cycle pays immediate dividends, and by performing a single round-trip instead of two, a factor of two in reduced latency is easily achieved.

We also evaluate the utility of the partial-write optimization in a more realistic benchmark setting. In addition to the PostMark benchmark, we also investigate the traffic savings on a debit-credit benchmark [28], a "TPC-B like" benchmark that is intended to model the workload on a database server that manages bank transactions [43].

Measurements reveal that the partial-write script reduces bandwidth considerably for both the PostMark and debit-credit benchmarks. For PostMark, total message traffic (in bytes) is reduced by roughly 54%. For the debit-credit benchmark, the savings are more dramatic, as a full 96% of the traffic is removed. The reason for these substantial reductions in network bandwidth is straightforward; both of these benchmarks perform many small writes, and by avoiding the page-level data transmissions required in a read-modify-write cycle, network load is considerably reduced.

4.5 Discussion

As we have seen in these case studies, the power to combine arbitrary operations at the server helps to overcome limitations in the exported RPC interface. As well as enabling clients to build customized performance optimizations, an important benefit of this approach arises on the server-side. With SRPC, the designer of a distributed service can focus on the simpler task of providing a few highly-tuned primitives, allowing clients to compose the primitives into the full scope of required functionality.

It should also be noted that SRPC can optimize the performance of network file systems in many ways other than the three case studies presented above. Specifically, whenever a client performs a predictable series of dependent operations on the server, and thus incurs multiple network round-trips, those operations can be combined into a single script. As an additional example, consider the case where a file system must ensure that a set of writes reach disk in a certain fixed order (*e.g.*, for crash recovery). In a traditional file system, the only way the client can ensure this ordering is to perform every write synchronously. We label this type of synchronous operation "false synchrony", because the client uses synchronous operations only to enforce order and *not* to ensure the operation has reached stable storage. Within the SRPC framework, false synchrony can be avoided, because the client could perform these writes asynchronously and still be guaranteed that the writes reach disk in the desired order.

5. FUNCTIONALITY ENHANCEMENTS

Traditional file servers implement a single, fixed protocol – a "one size fits all" solution that limits the functionality that clients can expect of the server. This inflexible approach wrongly assumes that a single protocol meets the requirements of all clients with respect to all files. For example, the consistency semantics that a client file system can implement are strongly constrained by the protocol interface exported by the server: a server that exports an NFS-like interface restricts all clients to a weak level of consistency, whereas a server that provides a stronger consistency model forces all clients to incur the consequent overhead.

SRPC allows clients to enhance the physical protocol provided by the server and thus implement enhanced *virtual protocols*. For example, SRPC can enable clients to implement more sophisticated consistency semantics on top of an NFS-like physical protocol; in our case studies, we implement both AFS and Sprite consistency semantics. In both cases, the ScFS interface remains the same and existing clients that use NFS-like semantics continue to operate smoothly. A key new feature of these examples is the demonstration that in our framework, state can be easily added to previously stateless protocols, such as NFS. Earlier work on Spritely NFS demonstrated some of the same capabilities, but did so by rewriting the server and client extensively [41]. Note that new consistency semantics are not the only possible functionality enhancement – for example, scripts could be utilized to implement an object-based disk interface on top of a block-based server, as hinted at in Section 4.4.

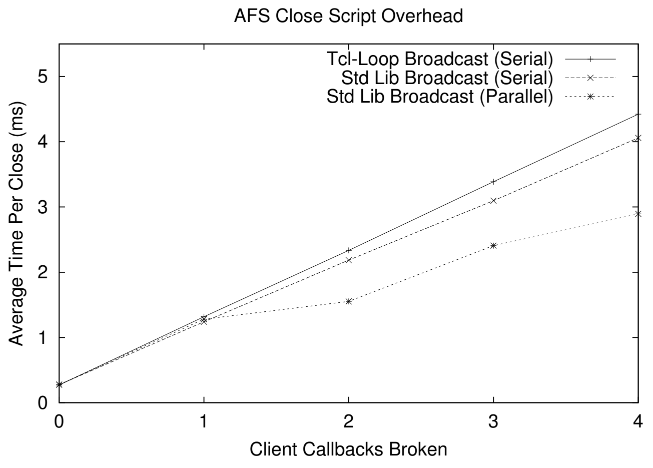


Figure 5: AFS Close Script Overhead. *The average cost of executing the* AFS CLOSE DIRTY *script is shown, as the number of callbacks that are broken increases.*

5.1 AFS Consistency

AFS [24] provides a write-on-close consistency model, in which clients see a consistent image of each file between an `open` and `close` operation. On a file open, the client installs a callback on the server and reads the entire file into its local disk cache. Subsequent reads and writes are performed on this local copy. When the file is closed, the client writes back the file, if modified, to the server. If another client stores a modified version of the file back to the server, all callbacks are broken and the clients invalidate their cached copies, forcing them to fetch the file from the server the next time they open the file. Since the server is actively involved in ensuring AFS consistency, a server that exports only the fixed NFS protocol cannot support AFS semantics.

With the SRPC infrastructure at the disk, implementing AFS consistency semantics is feasible, but servers must now be able to track state variables that live beyond the invocation of a single script. In AFS, the relevant state for each disk object is the callback list (*i.e.*, the list of client machines that currently have this file cached). Two new scripts are also required: AFS OPEN and AFS CLOSE DIRTY. On a file open, the client sends the AFS OPEN script which installs the client's IP address in the per-object callback list. The client then reads the object through the standard read interface and caches the file in its local disk. When a modified file is closed at the client, the client sends the AFS CLOSE DIRTY script to the server; this script loads the callback list associated with the object and invokes the `send_rpc_to_list` library routine to send a callback break to the listed clients.

Though the performance of a consistency model is difficult to measure, we present the time it takes to execute the AFS CLOSE DIRTY script, as the number of callbacks that must be broken is increased. Figure 5 plots the performance of three different implementations of the script. In the first, labeled "Tcl-Loop Broadcast (Serial)", the script itself sends a message synchronously to each of the clients on the callback list. In the second case, labeled "Std Lib Broadcast (Serial)", the Tcl script makes a single call to the C standard library routine, which then synchronously issues the RPC callbacks to all of the clients. Performance improves with this version due to the fewer number of crossings between the Tcl/C boundary. Finally, in the last case, labeled "Std Lib Broadcast (Parallel)", multiple threads are used within the standard library to issue call-

Experiment	**Avg. Read Cost (ms)**	**Overhead**
Direct Read	0.94	—
C	1.11	18.1%
Tcl (Cached)	1.43	52.1%

Table 1: Read During Write-Sharing Overhead. *The average cost of executing the* SPRITE READ CALLBACK *script is shown. The first row of the table, labeled "Direct Read", shows the cost of reading a 4 KB page synchronously, which serves as a lower bound on execution time. The second and third rows of the table show the cost of executing the C and Tcl implementations of the script, thus establishing the overhead that reads will experience during write-sharing. In each experiment, the read cost is calculated as the average of reads that occur during a large file copy.*

Experiment	**Avg. Update Cost (ms)**	**Overhead**
Base	0.94	—
C	1.04	10.6%
Tcl (Cached)	1.35	43.6%

Table 2: Directory Update Script Overhead. *The average cost of executing the directory update script is shown, in both C and Tcl implementations, and compared to a version without concurrency control. In each experiment, a new object ID is created, and then inserted into the directory. In both the C and Tcl implementations, only the new directory information is sent to the server; in contrast, the "Base" entry shows the performance of updating the directory directly, without any concurrency protocol. In that case, a 4 KB directory page is written to the server per update.*

back requests asynchronously; issuing the RPC callbacks in parallel improves performance slightly more. Thus, this experiment demonstrates the performance benefits of including the appropriate primitives in the standard library.

5.2 Sprite Consistency

Sprite [32] provides a stronger consistency model than AFS; these semantics, sometimes referred to as "perfect" consistency, are a close approximation of UNIX local file system semantics. In the Sprite model, clients can cache files as long as there is no write sharing of the file; write sharing occurs when more than one client has the file open and at least one of them has opened the file in write mode. When write-sharing occurs, Sprite turns off caching for the file and clients must send all reads directly to the server. Finding out if a file is currently write-shared requires the server to track the state of each file, which is not possible within an unmodified NFS-based system.

A major design concern when implementing Sprite semantics in ScFS is to ensure that reads and writes are not slowed down in the common case in which no write sharing occurs [26]. Since executing scripts has a negative performance impact, a script should not be sent as part of every read or write operation. To meet this goal, we developed a design in which each object has three associated state variables: a read callback list, which is a list of clients that have the file open (and cached) for reading; a writers list, which tracks the clients that currently have the file open for writing; and a flag designating whether or not this file is cacheable (*i.e.*, whether there is on-going write-sharing or not).

When a client does not know the state of a file or believes the file is non-cacheable, for each read request, the client sends a SPRITE READ CALLBACK script. The SPRITE READ CALLBACK script checks the cacheable flag; if the flag is set, the script adds the IP address of the client to the corresponding read callback list. The value of this flag is also returned to the client, so that the client can mark the state of the file in its local inode; if cacheable, then subsequent reads to the file are not scripted, but instead are issued as normal RPC read requests. Thus, when there is no write sharing, little overhead is paid for scripting.

When a client opens a file in write mode, the client sends a SPRITE WRITE OPEN script; this script sends an invalidate RPC to all members in the read callback list, marks the file non-cacheable, and adds its own IP address to the writers list for this object. If there is no other registered writer, the client marks the file as cacheable in its local inode and uses the in-memory cache to satisfy its read requests. When this client closes the file, it sends a SPRITE WRITE CLOSE script to the disk, which removes this client from the writers list, and marks the file as cacheable if the writers list is now empty. Once there are no registered writers, subsequent scripted-reads by other clients notice the cacheable status and cache the file.

The overhead of using scripts for reading is shown in Table 1. The "Direct Read" entry shows the average cost per read during a file copy, assuming no write-sharing. Thus, in the best case for this experiment, each read takes roughly 940 μs. The second entry shows the cost of executing the C version of the SPRITE READ CALLBACK script, which adds roughly an 18% overhead. Finally, the Tcl version of the same script runs approximately 52% more slowly than the non-scripted read. From this experiment, we see the importance of calling the SPRITE READ CALLBACK script only when necessary.

5.3 Discussion

In general, the ability to extend the interface on the server through scripts expands the types of functionality that client file systems can implement. However, as our case studies have shown, new functionality often requires that state can be associated with each object on the server. We briefly discuss two further examples where stateful scripts could be used to implement new ScFS functionality.

First, a client file system can implement fine-grained copy-on-write. In this system, state is required to track whether or not byte ranges within each object have yet been copied, and a new script is required for client reads and writes. A WRITE script transparently redirects the write to the copied version of the object, whereas a READ script directs the read to the appropriate version of the object. We imagine that optimizations similar to those used in the Sprite case study can be used to avoid invoking scripts on every read and write operation.

Second, clients could associate arbitrary type of meta-data with each file (*virtual meta-data*). One interesting piece of extensible meta-data would be a general access control list (ACL), which would provide more flexible sharing than the permission bits currently provided by our server. In such a system, on each read and write operation, the client would send a script to check the credentials of the user. However, to fully ensure that clients could not bypass this protection check by simply calling the existing RPC-based read and write calls, the permission bits would have to be disabled. This potential security hole illustrates a general principle: our scripts often assume the cooperation of a set of mutually-trusting clients. For example, if one client does not call the necessary scripts to enforce Sprite consistency semantics, other clients will not see the desired behavior.

6. SIMPLICITY ENHANCEMENTS

Since SRPC allows arbitrary operations to be grouped together and executed at the server, it greatly simplifies the implementation of atomic sets of operations that need to be isolated with respect to

Operation	Read & Get Attr (Sec. 4.2)	Get If Modified (Sec. 4.3)	Partial Write (Sec. 4.4)	AFS (Sec. 5.1) Open	AFS (Sec. 5.1) Close	Sprite (Sec. 5.2) Read Callback	Sprite (Sec. 5.2) Write Open	Sprite (Sec. 5.2) Write Close	Directory Update (Sec. 6.1)
Arithmetic ops	1-2	0-1	1			1			4-6
Control	1	1		1	1	2	3	2-3	1
List ops				1-2	4	0-2	4-6	2-5	
State mgmt				2	1	1-3	3-5	2-3	
Type checking	3-5	3-6	4	2	1	3	1		4
Library	3-6	3-4	3	1		3	1	1	7-8
Locks						2	2	2	2
Communication					1		1		
Native calls	1-2	1-2	2	1		1			2
Total lines	17	18	10	10	10	19	26	19	19

Table 3: Tcl Functional Breakdown. *The table categorizes static operation counts per script. For each script, the minimum and maximal path costs are shown. The left-most column presents the categories: arithmetic operations, control (`if` statements), list operations (create, add, iterate), state management (for stateful protocols), type checking, library (searches, copies, and other utility functions), locks (lock and unlock), communication (extra RPCs, beyond the mandated RPC reply), and native calls (calls to raw RPC-exported functions). The total at the bottom of each column lists the total number of lines in each script – note that this number may be lower or higher than the sum of the previous operations, as in some cases, statements are not accounted for separately (e.g., an end brace or some simple assignment statements), and in other cases, a line may consist of multiple statements (e.g., arguments which are actually math expressions). Each script is identified in the column header by its name as well as the section number in which it was described.*

concurrent operations from other clients. We demonstrate through a case study how SRPC can simplify the implementation of a seemingly complex distributed concurrency problem.

6.1 Concurrent Directory Updates

Given a disk object model in which directories are considered equivalent to any objects on disk, directory operations such as file create and delete translate into reads and writes at the server. Therefore, to create a file, a client first reads a directory page, inserts the directory entry pertaining to the file in a vacant slot, and writes the page back. Without proper concurrency control, simultaneous file creates within the same directory across different clients can lead to a lost create. Hence ScFS must ensure that the directory read-modify-write sequence is performed atomically.

With a traditional file server, a common way to ensure atomicity is through distributed locks, such that each client first acquires a lock for the directory object, then performs the read-modify-write, and finally unlocks the directory. Not only does this approach result in sub-optimal performance due to the multiple network round-trip operations required for the three phases, but more importantly, makes the overall system significantly more complex. Specifically, the server must now be able to track locks across multiple client machines and handle distributed failure scenarios such as a client crashing or momentarily losing network connectivity while it is holding a lock [42].

The SRPC framework greatly simplifies the implementation of atomic operations by co-locating on the server those operations that would otherwise be distributed across machines. In our concurrent directory update case study, each client sends a script to the disk that acquires an in-memory lock at the server, performs the read-modify-write, and then releases it. Thus, with SRPC, we can reduce a complex distributed concurrency problem to a much simpler challenge – that of ensuring mutual exclusion between threads in the server's address space.

Table 2 shows the performance of both the C and the Tcl implementation of the directory update, compared with the cost of simply sending a new directory page to the server (an approach that does not provide any concurrency control but represents the cost of simply writing a new page to perform the directory update). From the table, we can see the overhead of our simple concurrent directory update is satisfactory, providing new functionality without the difficulties that would be encountered in implementing a robust three-phase protocol.

6.2 Discussion

This case study has briefly illustrated that SRPC can simplify both client and server code. We believe that a centralized script at the server can help simplify functionality in many cases in which traditional distributed algorithms are required. For example, multi-update atomic transactions would be natural to provide within our scripting framework. Transactional capabilities would mandate additional functionality within the SRPC standard library, including the ability to roll back changes and perform crash recovery. In the future, we plan to investigate the utility of transactional support with the SRPC framework.

7. ANALYSIS OF TCL

In this section, we explore the costs of script execution within the Tcl environment. We provide a detailed accounting of each of the scripts that we have implemented, including functional and cost breakdowns. We end with a discussion of our findings.

7.1 Functional Breakdown

Table 3 shows the breakdown of Tcl commands for each of the scripts in our case studies, grouped into one of nine categories: arithmetic operations, control statements (such as `if` statements), list operations (a number of the scripts use lists as a basic data structure), state management routines (for saving and restoring long-lived server state), type checking (to ensure that no illegal memory dereferences occur), library routines (utility functions such as copies and string searches), locks, communication (beyond the single reply mandated by RPC), and native calls (the RPC-exported routines).

From the table, we make a number of general observations. From the last line in each column (the total number of lines per script), we observe that all of the scripts implement powerful functionality in a small amount of code. All scripts are in the "10s of lines of code" regime, ranging from a low of 10 up to a high of 26, with most in the 20-line range. At the low end are the AFS OPEN and AFS CLOSE DIRTY scripts, which simply manage the state required to track which clients have which files open, and the PARTIAL WRITE script, which performs a server-side read-modify-write in the expected manner. The most complex script is the SPRITE WRITE OPEN script, and even it is quite straight-forward – most of its lines consist of simple state and list management routines.

We also learn from the table how scripts are broken down into their constituent commands. Most scripts have relatively few control-flow decisions (the most complex, the SPRITE WRITE OPEN and SPRITE WRITE CLOSE scripts, each have three `if` statements), and thus are largely composed of straight-line code. For those scripts that require state (*i.e.*, the AFS and Sprite consistency scripts), much of their Tcl command count consists of retrieving the state, manipulating it (usually in the form of list operations), and then perhaps storing the state again. Finally, type checking and library operations comprise a substantial component of many of the scripts.

7.2 Cost Breakdown

Although a functional breakdown is instructive, without a time-based analysis, it would be difficult to pinpoint the location of bottlenecks. To garner such insight, we instrumented the scripts from each of our case studies, thus allowing us to collect detailed information as to where time is spent within each of the scripts. Figure 6 presents the results of our investigation.

From the figure, we make a number of observations. First, across all scripts, the invocation overhead accounts for a substantial portion of the time to execute scripts. This time consists of the C statements to set-up the relevant environment for each script, and then the call to `Tcl_Eval` to invoke the script. Though the percentage varies across scripts, the invocation overhead is fairly constant, varying between 150 to 200 μs. More detailed instrumentation reveals that most of this cost (roughly two-thirds) can be attributed to the `Tcl_Eval` call.

Second, type checking of buffer pointers and library commands combine to take a significant amount of time across many of the scripts. In the current system, each type check requires a call from Tcl into the C substructure, and all such calls are expensive. In future versions, we believe this cost could be reduced through batching of type checks. As for library commands, the routines that primarily are called are data copy routines, which are used to manipulate input parameters and construct return results. These overheads are difficult to avoid.

Third, additional communication is quite expensive, dwarfing most other costs. This effect is observed in the AFS CLOSE DIRTY script, which in this experiment breaks a single callback to a client who has a cached copy of the relevant file. The cost of communication within that script dominates all other costs.

Finally, native routines account for a reasonable amount of time across most of the scripts. Unlike the other components within each bar, the higher the percentage of the native portion of each bar, the better, as this portion represents direct calls to the underlying service, which is often the only "real" work that a script performs. For example, the native portion of the read-if-modified-since optimization consists of a `getattr` and a conditional `read`; all functions to copy data and perform type checking are pure overhead.

7.3 Discussion

We believe that the small code size demonstrated within these case studies is one of the best arguments for the scripting approach to extensibility. Small code segments are easier to write and maintain, and fewer lines of code implies fewer bugs, thus leading to more robust and reliable systems [11].

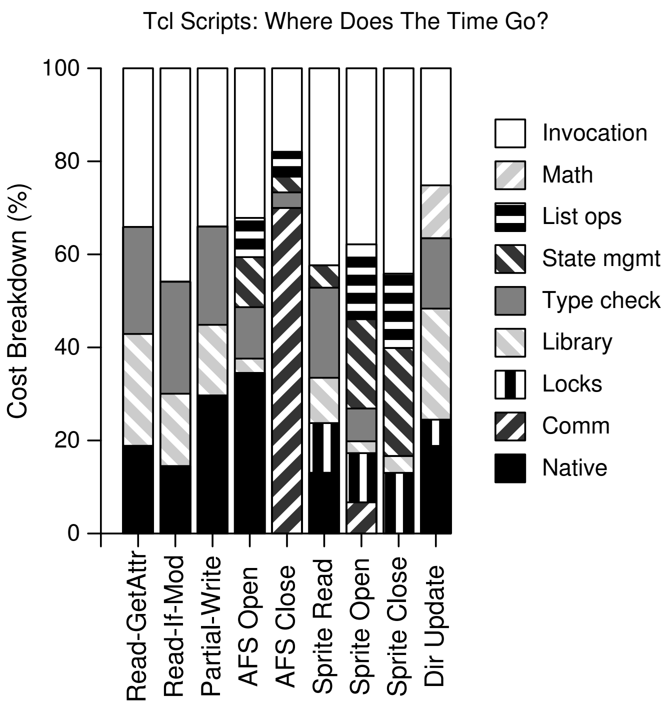


Figure 6: Tcl Cost Breakdown. *The graph depicts the percentage of time spent in each script in a typical scenario. Each bar represents a single script (as labeled along the x-axis), and is broken down into one of nine categories, based on the operational breakdown in Table 3. The only additional item in the breakdown is the "Invocation" cost, which is the time spent in getting from the C infrastructure into the Tcl script. Control statements have very low cost and therefore are not accounted for in the graph. The absolute execution times for each script are: 540 μs (Read-GetAttr), 304 μs (Read-If-Modified-Since), 590 μs (Partial-Write), 314 μs (AFS Open), 1299 μs (AFS Close), 377 μs (Sprite Read Callback), 393 μs (Sprite Write Open), 336 μs (Sprite Write Close), and 741 μs (Directory Update). Measurements reflect the average of 30 runs.*

However, our experience has brought forth some of the downsides of scripting as well. For example, "performance conscious" Tcl programming is much more difficult than the analogous process in C language environments – costs often arise from unexpected sources, making it difficult for programmers to optimize their code. For example, a simple math expression (`set x [expr 2+2]`) executes in roughly 5 μs on our Pentium-based platforms; a slight variant (`set x [expr 2 + 2]`), with a few extra spaces, executes in 30 μs, a factor of six slower! [3] In this specific case, the parsing of a single argument passed to the `expr` command is much faster than passing three separate arguments; in other cases, we have found that subtle differences in programming style can lead to non-trivial differences in performance. The more general problem illustrated herein is the difficulty of programming on top of a system as high-level as the Tcl interpreter; the more complex the virtual machine, the more difficult the process of programming for high-performance becomes.

[3] Sometimes two plus two makes six (times slower than expected).

8. RELATED WORK

8.1 Active Storage

Active storage has taken many forms in the existing literature, but almost all previous work has been in the context of new programming environments into which existing RPC-based services do not easily fit. The earliest work is found in the database literature [13], where researchers sought to exploit processing capability within each disk arm to increase database performance. More recent efforts in active storage were termed Active Disks and studied independently by Acharya [1] and Riedel *et al.* [37]. Acharya proposed a specialized "stream-based" programming model for parallel applications; in their model, applications are re-partitioned into host and disk portions, where each disk runs a "disklet" (*i.e.*, a small piece of Java code that filters data on a per-record granularity). Riedel *et al.* also studied parallel applications, focusing on scan-intensive codes; again, these applications must be partitioned across host and disks, where each disk runs some small portion of the code to filter requests and thus reduce total bandwidth to the host. Thus, these Active Disk systems are quite effective in supporting a variety of user-level parallel applications, but may not be appropriate for developing a more general distributed file service.

More recently, Amiri *et al.* introduced Abacus, an object-oriented framework for developing active storage systems [2]. Abacus is most similar to SRPC in that the authors developed a distributed object storage system on top of Abacus and demonstrated performance benefits. However, Abacus differs in that the distributed object store is built from scratch within their distributed object environment and most code exists only at the user-level; thus, Abacus is not well-integrated into the kernel (*e.g.*, one can mount an Abacus-based file system, but system calls are redirected from inside the kernel into a user-level proxy, which can be inefficient [9]). One of the main strengths of the Abacus approach is that work is dynamically migrated to the client or the server depending on system and workload characteristics. A similar adaptive framework could be utilized by SRPC as well.

8.2 Extensibility

SRPC is also related to a long line of work in extensible systems, pioneered by systems such as SPIN [5], Exokernel [15], and VINO [39]. As all of those systems sought to enable extensibility for operating systems, we seek to enable extensibility for RPC-based services. Many of the lessons learned in those systems apply to SRPC; for example, some of the techniques used in VINO to survive misbehaving kernel extensions are directly applicable to our framework.

Slice is a virtual file service that extends services on only the client side via interposition [3]. By introducing client-side packet filters, Slice can build a virtual file service on top of existing protocols such as NFS, and does so transparently to file system clients. Thus, Slice's interposition and SRPC server-side scripting are complimentary approaches, with Slice adding "activity" to the client, and SRPC adding it to the server.

Others have suggested the utility of scripting languages within extensible systems. The most direct example of this is found within the the μChoices operating system [8]. Therein, the authors suggest the use of a Tcl-like scripting language to specify OS extensions and argue that the flexibility and safety provided by interpreted languages outweigh the potential performance loss; this hypothesis is one of the topics we investigate in this paper. Similarly, the Swarm scalable storage system sends Tcl scripts to servers in order to read and write data [23]. However, as the authors state, Swarm makes little use of this feature other than for debugging.

8.3 RPC

In the realm of RPC, we are not aware of a system that is highly similar to SRPC. Most recent work in this area concentrates on increasing the performance or flexibility of the RPC substrate, or reducing code size of automatically-generated stubs [14, 20, 34]. An excellent example is found in the work on Flick, a flexible infrastructure for building an optimizing RPC layer [14]. The main goal of Flick is to separate efficient stub generation from both the specific interface-definition language (IDL) and underlying communication layer. Our work on Scriptable RPC could likely be extended into the Flick framework.

9. CONCLUSIONS

Given current technology trends, we expect that the core building blocks of future systems will contain significant processing capabilities. Thus, it is imperative that future distributed services have the ability to effectively leverage these active components. In this paper, we have introduced SRPC, a scriptable RPC layer that enables system developers to migrate existing distributed services onto active servers.

We have designed SRPC to meet three goals that are essential in this environment. First, SRPC is designed to provide a smooth migration path for existing distributed services from a traditional RPC-based server to an active server. This goal is accomplished by making the process of developing an SRPC-based service similar to that of developing a service within the traditional RPC paradigm, automating as many of the steps as possible with an enhanced IDL compiler. Second, SRPC is engineered for high performance; SRPC caches scripts, provides support for concurrent script execution, and allows key operations to be implemented directly in C for efficiency. Third, SRPC targets safe extensibility; using Tcl as the base scripting language is a crucial factor in meeting this goal.

Through a number of case studies, we have demonstrated three general benefits of ScFS over a traditional RPC-based file system. First, we have shown that SRPC can improve client performance; scripting allows the client to merge operations with dependencies into a single operation between the client and the disk, thereby reducing the number of network round-trips. Second, we have demonstrated that SRPC enables new functionality to be easily integrated into the file system; this is useful not only in those cases where the original designers did not foresee the benefits of this functionality, but also for those cases where different clients desire different functionality for different files. Finally, we have shown that scripts permit operations to be co-located at the disk instead of being distributed across multiple clients (*e.g.*, acquiring and releasing locks), thereby avoiding complex code for crash recovery or distributed failure scenarios.

In our evaluation of Tcl, we find that Tcl enables the development of short but powerful scripts – even the most complex of our case studies required less than thirty lines of Tcl code. We also find that the performance of Tcl is much better than in the past and that Tcl is appropriate for many file system extensions. However, in some scenarios, higher performance is required, and a specialized domain-specific language for active disks may be worth investigating. One desirable feature of such a language would be a predictable cost model, allowing programmers to optimize their code in a direct and obvious manner.

In the future, we believe it would be interesting to examine SRPC in the context of multiple network-attached disks instead of a single server. With an active framework, disks can communicate and cooperate with one another directly, perhaps implementing advanced features such as snap-shots [27] or lazy redundancy [12]. The key challenge for such a system is to provide the proper primitives for distributed coordination among server-side scripts, hence removing the burden of implementing complex distributed systems protocols. Perhaps the ideal storage system of the future is as simple as this: a collection of SRPC-enabled disks and a few base primitives for distributed computation in the SRPC standard library, with all higher-level file-system functionality built on top of a flexible and efficient scripting substrate.

Acknowledgments

We would like to thank Venkateshwaran Venkataramani for his work on this project in its early stages, and his skepticism which led to the refinement of many of the ideas herein. We would also like to thank the members of the WiND research group for their input on the ideas and presentation of this paper, and the CS-838 class for their lively feedback. Finally, we thank the anonymous reviewers for their excellent and thoughtful suggestions, which have greatly improved the content of this paper. This work is sponsored by NSF CCR-0092840, CCR-0098274, NGS-0103670, CCR-0133456, ITR-0086044, and the Wisconsin Alumni Research Foundation.

10. REFERENCES

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, CA, October 1998.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 307–322, June 2000.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. *Transactions on Computer Systems (TOCS)*, 20(1), February 2002.

[4] T. Berners-Lee, R. T. Fielding, H. F. Nielsen, J. Gettys, and J. Mogul. Hypertext Transfer Protocol — HTTP/1.1. Technical Report 2068, Internet Engineering Task Force, January 1997.

[5] B. N. Bershad, S. Savage, E. G. S. Przemyslaw Pardyak, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet—A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29–38, February 1995.

[8] R. H. Campbell and S. M. Tan. μChoices: An Object-Oriented Multimedia Operating System. In *In Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, Orcas Island, WA, May 1995.

[9] J. B. Chen and B. Bershad. The Impact of Operating System Structure on Memory System Performance. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 120–133, Asheville, NC, December 1993.

[10] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 267–280, November 14–17 1994.

[11] A. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[12] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, CA, June 2002.

[13] D. J. DeWitt and P. B. Hawthorn. A Performance Evaluation of Data Base Machine Architectures. In *Proceedings of the Seventh Annual Conference Very Large Data Bases (VLDB '81)*, pages 199–214, 1981.

[14] E. Eide, K. Frei, B. Ford, J. Lepreau, and G. Lindstrom. Flick: A Flexible, Optimizing IDL Compiler. In *PLDI '97*, Las Vegas, NV, June 1997.

[15] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[16] M. Fiuczynski, R. Martin, B. Bershad, and D. Culler. SPINE: An operating system for intelligent network adapters. Technical Report TR-98-08-01, University of Washington, Department of Computer Science and Engineering, August 1998.

[17] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[18] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284, Seattle, WA, June 1997.

[19] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie-Mellon University, 1997.

[20] A. Gokhale and D. C. Schmidt. Measuring the Performance of the CORBA Internet Inter-ORB Protocol over ATM. Technical Report WUCS-97-09, Washington University at St. Louis, 1997.

[21] J. Gray. Storage Bricks Have Arrived. Invited Talk at the First USENIX Conference on File And Storage Technologies (FAST '02), 2002.

[22] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, October 2000.

[23] J. H. Hartman, I. Murdock, and T. Spalink. The Swarm Scalable Storage System. In *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, Texas, June 1999.

[24] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems (TOCS)*, 6(1), February 1988.

[25] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[26] J. Kistler and M. Satyanarayanan. Disconnected Operation in

the Coda File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1), February 1992.

[27] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 84–92, Cambridge, MA, October 1996.

[28] D. E. Lowell and P. M. Chen. Free Transactions With Rio Vista. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, Saint-Malo, France, October 1997.

[29] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A Modular Reconfigurable Architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[30] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. In *Proceedings of the 24rd Annual International Symposium on Computer Architecture*, pages 85–97, Denver, Colorado, June 2–4, 1997. ACM SIGARCH and IEEE Computer Society TCCA.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A Low-Bandwidth Network File System. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP-01)*, pages 174–187, Banff, Canada, October 2001.

[32] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite Network File System. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[33] C. Nester, M. Philippsen, and B. Haumacher. A More Efficient RMI for Java. In *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, California, June 1999.

[34] S. O'Malley, T. Proebsting, and A. B. Montz. USC: A Universal Stub Compiler. In *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM '94)*, London, UK, August 1994.

[35] J. K. Ousterhout. Tcl: An Embedable Command Language. In *Proceedings of the 1990 USENIX Association Winter Conference*, 1990.

[36] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. Intelligent RAM (IRAM): Chips That Remember And Compute. In *1997 IEEE International Solid-State Circuits Conference*, San Francisco, CA, February 1997.

[37] E. Riedel, G. A. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proc. of the 24th International Conference on Very large Databases (VLDB '98)*, August 1998.

[38] T. H. Romer, D. Lee, G. M. Voelker, A. Wolman, W. A. Wong, J.-L. Baer, B. N. Bershad, and H. M. Levy. The structure and performance of interpreters. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 150–159, Cambridge, Massachusetts, October 1–5, 1996. ACM SIGARCH, SIGOPS, and SIGPLAN.

[39] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

[40] C. Small and M. Seltzer. A Comparison of OS Extension Technologies. In *Proceedings of the 1996 USENIX Annual Technical Conference*, January 1996.

[41] V. Srinivasan and J. C. Mogul. Spritely NFS: Experiments with cache-consistency protocols. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 45–57. ACM, December 1989. Order no. 534890.

[42] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[43] Transaction Processing Council. TPC Benchmark B Standard Specification, Revision 3.2. Technical Report, 1990.

[44] T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-net: A user-level network interface for parallel and distributed computing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 40–53, Copper Mountain Resort, CO, USA, 1995.

[45] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[46] M. Welsh and D. Culler. Achieving Robust, Scalable Cluster I/O in Java. In *LCR2000: Fifth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, Rochester, NY, May 2000.