RPC Chains: Efficient Client-Server Communication in Geodistributed Systems

Yee Jiun Song[1,2] Marcos K. Aguilera[1] Ramakrishna Kotla[1] Dahlia Malkhi[1]
[1] *Microsoft Research Silicon Valley* [2] *Cornell University*

Abstract

We propose the RPC chain, a simple but powerful communication primitive that allows an application to reduce the performance effects of wide-area links on enterprise and data center applications that span multiple sites. This primitive chains together multiple RPC invocations so that the computation can flow from one server to the next without involving the client every time. We demonstrate that RPC chains can significantly reduce end-to-end latency and network bandwidth in a storage application and a web application.

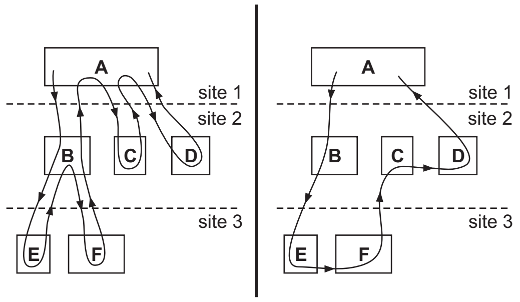

Figure 1: (**Left**) Standard RPCs. (**Right**) RPC chain.

1 Introduction

Distributed enterprise applications, such as web applications, are often built from more basic services, such as storage services, database management systems, authentication and configuration services, and services for interfacing with external components (e.g., credit card processing, banking, vendors, etc). As systems become larger, more complex, and more ubiquitous, there is a corresponding increase in the number, diversity, and geographical dispersion of the remote services that they use. For instance, Hotmail and Live Messenger share an address book service and an authentication service; there are also services specialized for each application, say, for email storage or virus scanning. These services are heterogeneous; they are often developed by different teams and are *geo-distributed*, running in different parts of the world.

Geo-distribution provides many benefits: high availability, disaster tolerance, locality, and ability to scale beyond one data center or site. However, the thin and slow links connecting different sites pose challenges, especially in an enterprise setting, where applications have strict performance requirements. For instance, web applications should ideally respond within one second [13].

The most common primitives for inter-service communication are remote procedure calls (RPC's) or RPC-like mechanisms. RPC's can impose undesirable communication patterns and overheads when a client needs to make multiple calls to servers. This is because RPC's impose communication of the form $A{-}B{-}A$ (A calls B which returns to A) even though this pattern may not be optimal. For example, in Figure 1 left, a client A in site 1 uses RPC's to consecutively call servers B, C, and D in site 2. Server B, in turn, calls servers E and F in site 3. The use of RPC's forces the execution to return to A and B multiple times, causing 10 crossings of inter-site links

We propose a simple but more general communication primitive called a *Chain of Remote Procedure Calls*, or simply *RPC chain*, which allows a client to call multiple servers in succession ($A{-}B_1{-}B_2{-}\cdots{-}A$), where the request flows from server to server without involving the client every time. The result is a much improved communication pattern, with fewer communication hops, lower end-to-end latency, and often lower bandwidth consumption. In Figure 1 right, we see how an RPC chain reduces the number of inter-site crossings to 4. The example in this figure is representative of a web mail application, where host A is a web server that retrieves a message from an email server B, then retrieves an associated calendar entry from a calendar service C, and finally retrieves relevant ads from an ad server D.

The key idea of RPC chains is to embed the chaining logic as part of the RPC call. This logic can be a generic function, constrained by some simple isolation mechanisms. RPC chains have three important features:

- *(1) Server modularity.* What made RPC's so successful is the clean decoupling of server code, which allows servers to be developed independently of each other and the client. RPC chains preserve this attribute, even allowing existing legacy RPC's to be part of a chain through simple wrappers.
- *(2) Chain composability.* If a server in the chain itself wishes to call another server, this nested call can be simply added to the chain in flux. In Figure 1, when client A starts the chain, it intends to call only servers B, C, and D. But server B wants to call servers E and F, and so it adds them to the chain.
- *(3) Chain dynamicity.* The services that a host calls need not be defined a priori; they can vary dynamically during execution. In the left figure, the fact that client A calls servers C and D need not be known before A calls server B; it can depend on the result returned by B. For example, an error condition may cause a chain to end immediately instead of continuing on to the next server.

We demonstrate RPC chains through a storage and a web application. For the storage application, we show how a storage server can be enabled to use RPC chains, and we give a simple use in which a client can copy data between servers without having to handle the data itself. This speeds up the copying and saves significant bandwidth. For the web application, we implement a simple web mail service that uses chains to reduce the overheads of an ad server.

The paper is organized as follows. We explain the setting for RPC chains in Section 2. Section 3 covers the design of RPC chains and Section 4 covers applications. We evaluate RPC chains in Section 5, and we explain their limitations in Section 6. A discussion follows in Section 7. We discuss related work in Section 8 and we conclude the paper in Section 9.

2 Setting

We consider enterprise systems that span geographically-diverse sites, where each site is a local area network. Sites are connected to each other through thinner and slower wide area links. Wide-area links can be made faster by improving the underlying network, and lots of progress has been made here, but this progress is hindered by economic barriers (e.g., legacy infrastructure), technological obstacles (e.g., switching speeds), and fundamental physical limitations (e.g., speed of light). Thus, the large discrepancy between the performance of local and wide-area links will continue.

Unlike the Internet as a whole, enterprise systems operate in a trusted environment with a single administrative domain and experience little churn. These systems may contain a wide range of services, often developed by many different teams, including general services for storage, database management, authentication, and directories, as well as application-specific services, such as email spam detection, address book management, and advertising. These services are often accessed using RPC's, which we broadly define as a mechanism in which a client sends a request to a server and the server sends back a reply. This definition includes many types of client-server interactions, such as the interactions in CORBA, COM, REST, SOAP, etc.

In enterprise environments, application developers are not malicious though some level of isolation is desirable so that a problem in one application or service does not affect others.

3 Design

We now explain the design of RPC chains, starting with the basic mechanism for chaining RPC's in Section 3.1. The code that chains successive RPC's is stored in a repository, explained in Section 3.2. In Section 3.3, we cover the state that is needed during the chain execution. We then discuss composition of chains in Section 3.4, legacy servers in Section 3.5, isolation in Section 3.6, debugging in Section 3.7, exceptions in Section 3.8, failures in Section 3.9, and chain splitting in Section 3.10.

3.1 Main mechanism

Servers provide services in the form of *service functions*, which is the general term we use for remote procedures, remote methods, or any other processing units at servers. An RPC chain calls a sequence of service functions, possibly at different servers. Service functions are connected together via *chaining functions*, which specify the next service function to execute in a chain (see Figure 2 top). Chaining functions are provided by the client and executed at the server. They can be arbitrary C# methods with the restriction that they be *stand-alone* code, that is, code which does not refer to non-local variables and functions, so that they can be compiled by themselves.

We chose this general form of chaining for two reasons. First, we want to allow the chain to unfold dynamically, so that the choice of next hop depends on what happens earlier in the chain. For example, an error at a service function could shorten a chain. Second, we wanted to support server modularity, so that services and client applications can be developed independently. Thus, a server may not produce output that is immediately ready for another server, in the way intended by the client's application. One may need to convert formats, reorder parameters, combine them, or even combine the outputs from several servers in the chain. For example, an NFS

```
// service function
object sf(object parmlist)
  // parmlist:  parameter list
// chaining function
nexthop cf(object state, object result)
  // state:  from client or earlier parts of chain
  // result:  from last preceding service function
  // returns next chain hop:
  //        (server, sf_name, parmlist,
  //        cf_name, state)
```

```
chain_id start_chain(machine_t server,
        string sf_name, object parmlist,
        string cf_name, object state)
```

Figure 2: **(Top)** Signature of a service function (sf) and chaining function (cf). **(Bottom)** Signature of function that launches an RPC chain.

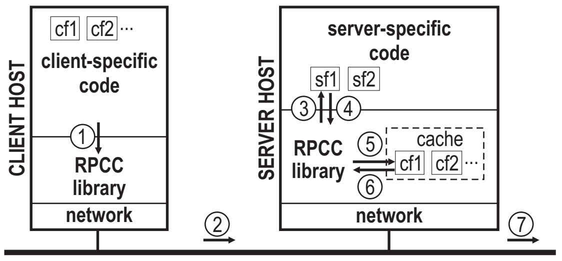

Figure 3: Execution of an RPC chain (see explanatory text in Section 3.1). RPCC stands for RPC chain.

server does not output data in the format expected by a SQL server: one needs glue that will convert the output, choose the tables, and add the appropriate SQL wrapper, according to application needs. Chaining functions provide this glue. We initially considered a simpler alternative to chaining functions, in which a client just provides a static list of servers to call, but this design does not address the issues above. We also note that it is easy to translate a static server list into the appropriate chaining functions (one could even write a programmer tool that automatically does that), so our design includes static lists as a special case.

Figure 3 shows how an RPC chain executes. (1) A client calls our RPCC (RPC chain) library, specifying a server, a reference to a service function sf_1 at that server, its parameters, and a chaining function cf_1. (2) This information is then sent to the chosen server. (3) The server executes service function sf_1, which (4) returns a result. (5) This result is passed to the chaining function cf_1, which then (6) returns the next server, service function, and chaining function, and (7) the chain continues.

For example, suppose client A wants to call service functions sf_B, sf_C, sf_D at servers B, C, and D, in this order. To do so, the client specifies a reference to sf_B and a chaining function cf_1. cf_1 causes a call to sf_C at server C with a chaining function cf_2, which in turn causes a call to sf_D at server D with a chaining function cf_3, which causes the final result to be returned to the client A.

3.2 Chaining function repository

Chaining functions are provided by clients but executed at servers. To save bandwidth, in our implementation the client does not send the actual code to the server. Rather, the client uploads the code to a repository, and sends a reference to the server; the server downloads the code from the repository and caches it for subsequent use. The repository stores chaining functions in source code format, and servers compile the code at runtime using the reflection capabilities of .NET/C# (Java has similar capabilities).

We store source code because it introduces fewer dependencies, is more robust (binary formats change more frequently), and simplifies debugging. Because the cost of runtime compilation can be significant (≈50 ms, see Section 5.2.1), servers cache the compiled code, not the source code, to avoid repeated compilations.

When the chaining function is very small, it could be transmitted by the client with the RPC chain, so that the server does not have to contact the repository. Our implementation presently does not support this option.

3.3 Parameters and state

A chaining function is client logic that may depend on run-time variables, tables, or other state from the client or earlier parts of the chain. This state needs to be passed along the chain, and ideally it should be small, otherwise its transmission cost can outweigh the benefits of an RPC chain (see Section 5.2.2). We represent the state as a set of name-value pairs, which is passed as a parameter to the chaining function (see Figure 2).

The output of each service function is also passed as a parameter to the subsequent chaining function. For example, in our storage copy application (Section 4.1), the first service function reads a file, and the chaining function uses the result as input to the next service function, which writes to a file on a different server. In our email application, a service function reads an email message, and the chaining function adds the message to the state of the next chaining function, so that the message is passed along the chain back to the chain originator (a mail web server).

3.4 Nesting and composition

RPC chains can be nested: a service function in a chain may itself start a sub-chain. For example, the main chain could call a storage service, which then needs to call a

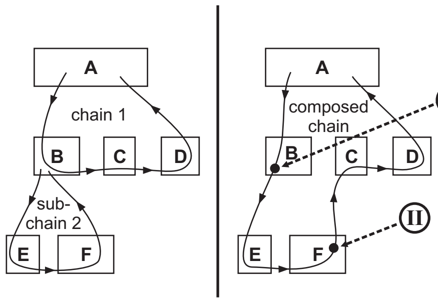

Figure 4: Composition of nested chains. **(Left)** The main chain 1 and a sub-chain 2. **(Right)** Result and manner of composing chains. (I) B starts a sub-chain, causing the RPCC library to push the $B \rightarrow C$ chaining function and its state parameter into a stack. (II) Chaining function at F returns an indication that the chain ended and the result that B is supposed to produce. This causes the RPCC library to pop from the stack, obtaining the $B \rightarrow C$ chaining function and its state parameter. It then calls this chaining function with the result and state. The chain now continues at C.

replica. We implement nesting so that a nested chain can be adjoined to an existing chain, as shown in Figure 4. Note the difference between starting a chain going from B to E, and moving to the next host in a chain going from C to D: the former occurs when the *service function* at B starts a new chain, while the latter occurs when the *chaining function* at C calls the next node in the chain. This distinction is important because the service function at B represents a native procedure at the service, while a chaining function at C represents logic coming from A. At E, the chaining function that calls F represents logic coming from B.

To compose a chain with its sub-chain, the chaining function of the parent chain needs to be invoked when a sub-chain ends (to continue the parent chain). Accordingly, when a host starts a sub-chain, the RPCC library saves the chaining function and its state parameter, and passes them along the sub-chain. The sub-chain ends when its chaining function returns null in `nexthop.server`, and a result in `nexthop.state` (this is the result that the host originating the sub-chain must produce for the parent chain). When that happens, the RPCC library calls the saved chaining function with the saved state and `nexthop.state`. Note that a chain and a sub-chain need not be aware of each other for composition.

To allow multiple levels of nesting, we use a *chain stack* that stores the saved chaining function and its state for each level of composition. The stack is popped as each sub-chain ends.

3.5 Handling legacy RPC services

RPC chains support legacy services that have standard RPC interfaces. For that, we use a simple wrapper module, installed at the legacy RPC server, which includes the RPCC library and exposes the legacy remote procedures as service functions.

Each service function passes requests and responses to and from the corresponding legacy remote procedure. Because the service function calls the legacy remote procedure locally through the RPC's standard network interface (e.g., TCP), the legacy server will see all requests as coming from the local machine, and this can affect network address-based server access control policies. (This is not a problem if access control is based on internal RPC authenticators, such as signatures or tokens, which can be passed on by the wrapper.)

One solution is to re-implement the access control mechanism at the wrapper, but this is application-specific. A better solution is for the wrapper to fake the network address of its requests and capture the remote procedure's output before it is placed on the network.

3.6 Isolation

Chaining functions are pieces of client code running at servers. Even though clients are trustworthy in the environment we consider, they are still prone to buffer overruns, crashes, and other problems. Thus, chaining functions are sandboxed to provide isolation, so that client code cannot crash or otherwise adversely affect the server on which it runs.

We need two types of isolation: (1) restricting access to sensitive functions, such as file and network I/O and privileged operating system calls, and (2) restricting excessive consumption of resources (CPU and memory).

We achieve (1) through direct support by .NET/C# of access restrictions to file I/O, system and environment variables, registry, clipboard, sockets, and other sensitive functions (Java has similar capabilities). This is accomplished by placing descriptive annotations, called *attributes*, in the source code of chaining functions when they are compiled at run-time.

We achieve (2) by monitoring CPU and memory utilization and checking that they are within preset values. The appropriate values are a matter of policy at the server, but for the short-lived type of executions that we target with RPC chains, chaining functions should consume at most a few CPU seconds and hundreds of megabytes of memory, even in the most extreme cases.

If a chaining function violates restrictions on access or resource consumption, an RPC chain exception is thrown according to the mechanism in Section 3.8.

Another way to isolate chaining functions is to use a chaining proxy (Section 7.3).

3.7 Debugging and profiling

A very useful debugging tool for traditional applications is "printf", which allows an application to display messages on the console. We provide an analogous facility for RPC chain applications: a *virtual console*, where nodes in the chain can log debugging information. The contents of the virtual console are sent with the chain, and eventually reach the client, which can then dump the contents to a real console or file. The virtual console can also be used to gather profiling information for each step in the chain and be aggregated at the client.

Even with "printf", debugging RPC chains can be hard, because it involves distributed execution over multiple machines. We can reduce this problem to the simpler problem of debugging RPC-based code by running RPC chains in a special *interactive mode*. The key observation is that chaining functions are *portable code* that can be executed at any machine. In interactive mode, chaining functions always execute at the client instead of the servers. To accomplish this, after each service function returns, the RPCC library sends its result back to the client, which then applies the chaining function to continue the chain from there. A chain executed in interactive mode looks like a series of RPC calls. By running the client in an interactive debugger, the developer can control the execution of the chain and inspect the outputs of service and chaining functions at each step.

3.8 Exceptions

An RPC chain may encounter exceptional conditions while it is executing: (1) the next server in the chain can be down, (2) the chaining function repository can be down, or (3) the state passed to the chaining function can be missing vital information due to a bug. All of these will result in an exception, either at the RPCC library in cases (1) and (2), or at a chaining function in case (3). (Service functions do not throw exceptions; they simply return an error to the caller.)

Who should handle such exceptions? One possibility is to handle them locally, by having the client send exception handling code as part of the chain. Doing this requires sending all the state that the handling code needs, which complicates the application design. Instead, we choose a less efficient but simpler alternative (since exceptions are the rare case). We simply propagate exceptions back to the client that started the chain. The client receives the exception name, its parameters, and the path of hosts that the chain has traversed thus far. (If the client crashes, the exception becomes moot and is ignored.)

In the case of nested chains, the exception propagates first to the host that started the current sub-chain. If that host does not catch the exception, it continues propagating to the host that started the parent chain, until it gets to the client. For example, in Figure 4 right, if E throws an exception (say, because it could not contact F), the exception goes to B, the node that created the sub-chain. This is a natural choice because B understands the logic of the sub-chain that it created, and so it may know how to recover from the exception. If B does not catch the exception, it is propagated to A.

3.9 Broken chains

The crash of a host while it executes an RPC chain results in a *broken chain*. In this section, we describe the broken chain detection and recovery mechanisms.

Detection. We detect a broken chain using a simple end-to-end timeout mechanism at the client called *chain heartbeats*: a chain periodically sends an alive message to the client that created it, say every 3 seconds, and the client uses a conservative timeout of 6 seconds. If there are sub-chains, only the top-level creator gets the heartbeats. Heartbeats carry a unique chain identifier, a pair consisting of the client name and a timestamp, so that the client knows to which chain it refers.

We achieve the periodic sending through a time-to-heartbeat timer, which is sent with the chain, and it is decremented by each node according to its processing time, until it reaches 0, the time to send a heartbeat. Synchronized clocks are not needed to decrement the timer; we only need clocks that run at approximately the same speed as real time. Since we do not know link delays, we assume a conservative value of 200 ms and decrement the time-to-heartbeat timer by this amount for every network hop. This assumption may be violated when if there is congestion and dropped packets, resulting in a premature timeout (false positive). However, the impact of false positives is small because of our recovery mechanism, explained next.

Recovery. To recover from a broken chain, the client simply retransmits the request. Like standard remote procedures, we make chains idempotent by including a chain-id with each chain, and briefly caching the results of service functions and chaining functions at each server. If a server sees the same chain-id, it uses the cached results for the service and chaining functions. The chain can continue in this fashion up to the host where the chain previously broke. At that host, if the "next" host is still down, an exception is thrown. Alternatively, a fail-over mechanism that calls a backup server can be implemented by using logical server names which are mapped to a backup when the primary fails. This is similar to the mechanisms used to fail over standard RPC's.

Upon a second timeout, a client executes the RPC chain in *interactive mode* (as in Section 3.7), to determine exactly at which node the chain stopped, and returns an error to the application.

3.10 Splitting chains

For performance reasons, it may be desirable to split a chain to allow parallel execution. The decision to split a chain should be made with consideration of the added complexity, as concurrent computations are always harder to understand, design, debug, and maintain compared to sequential computations. Although our applications do not use splitting chains, we now explain how such chains can be implemented.

Split. We modify chaining functions so that they can return more than one *nexthop* parameter. The RPCC library calls each nexthop concurrently, resulting in the several split-chains. Each chain has an id comprised of the id of the parent plus a counter. For example, if there is a 3-way split of chain 74, the split-chains will have ids 74.1, 74.2, and 74.3. Each of these split chains can in turn be split again, and result in split-chains with increasingly long ids. For example, if split-chain 74.1 splits into two, the resultant split-chains will have ids 74.1.1 and 74.1.2. We note for future reference that each split-chain knows how many siblings it has (this information is passed on to the split-chains when the chain splits).

Broken split chains. Recall that we use an end-to-end mechanism to handle broken chains (Section 3.9) via a chain heartbeat. When a chain splits, we also split the heartbeats: each split-chain sends its own heartbeat (with the split-chain id) and the client will be content only if it periodically sees the heartbeat from all the split-chains. The heartbeat messages indicate the number of sibling split-chains, so that the client knows how many to expect. If a split-chain is missing, the client starts the chain again (even if other split-chains are still running, this does not cause a problem because of idempotency).

Merge. To merge split-chains, a *merge host* collects the results of each split-chain and invokes a *merge function* to continue the chain. The merge host and function are chosen when the chain splits (they are returned by the chaining function causing the split). The merge host can be any host; a good choice is the next host in the chain. The merge host awaits outcomes from all split-chains before calling the merge function, which takes the vector of results and returns *nexthop*, specifying the next service function and chaining function to call.

After split-chains complete (i.e., reach the merge host), the parent chain will continue and resume its heartbeats. However, split-chains do not necessarily complete at the same time, so there may be a period from when the first split-chain completes until the parent chain resumes. During this period the merge host sends heartbeats on behalf of the completed split-chains, so that the client does not time out.

Crash garbage. When there are crashes in the system, the merge host may end up with the outcome of stale split-chains. This garbage can be discarded after a timeout: as we mentioned, RPC chains are intended for short-lived computations, so we propose a timeout of a minute. Note that if a slow system causes a running chain to be garbage collected, the client will recover after it times out.

4 Applications

To demonstrate RPC chains, we apply and evaluate them in two important enterprise applications: a storage application (Section 4.1) and a web application (Section 4.2).

4.1 Storage applications

Storage services generally provide two basic functions, *read* and *write*, based on keys, file names, object id's, or other identifiers. While this generic interface is suitable for many applications, its low-level nature sometimes forces bad data access patterns on applications. For instance, if a client wants to copy a large object from one storage server to another, the client must read the object from one server and write it to the other, causing all the data to go through the client. If the client is separated from the storage servers by a high latency or low bandwidth connection, this copying could be very slow.

One solution is to modify the storage service on a case-by-case basis for different operations and different settings. For example, the Amazon S3 storage service recently added a new copy operation to its interface [2], so that an end user can efficiently copy her data between data centers in the US and Europe, without having to transfer data through her machine. Although such application-specific interfaces can be beneficial, they are specific to particular operations and do not mitigate adverse communication patterns in other settings.

RPC chains provide a more general solution: they not only enable the direct copying of data from one server to another (through a simple chain that reads and then writes), but also enable broader uses. To demonstrate this idea, we layered RPC chains over a legacy NFS v3 storage server, as explained in Section 3.5. (We could have used other types of storage, such as an object store.) We then implemented a simple chain to copy data without passing through the client.

We also show a more sophisticated application of chains by implementing a primary-backup replication of

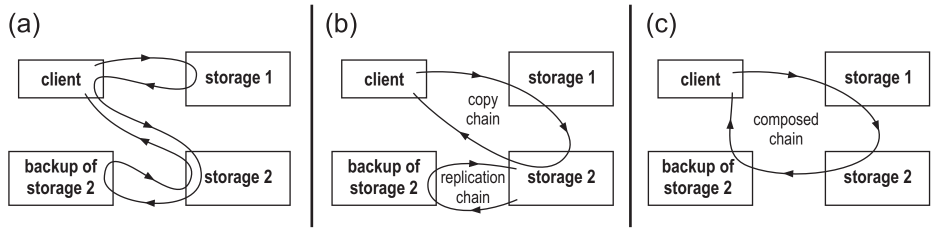

Figure 5: (a) Copying data from storage server 1 to a replicated storage server 2 without RPC chains. The client reads from storage 1 and writes to storage 2; when this happens, storage 2 writes to a backup server. (b) Using a chain to copy data and a chain to replicate data (composition disabled). (c) Composing the chains. The chains are not aware of each other but the RPCC library can combine them.

the storage server: when the primary receives a write request, it creates a chain to apply the request on a backup server. Because replication is done through chains, it can be composed with other chains. This is illustrated in Figure 5(b), which shows a setup with two storage servers, the second of which is replicated, and a user who wants to copy data from the first to the second server. Two chains are created as a result of this request: a chain that the client launches for copying, and another that the second storage server launches for replication. The RPCC library allows these two chains to be composed together, as shown in Figure 5(c). We report on quantitative benefits of our approach in Section 5.3.

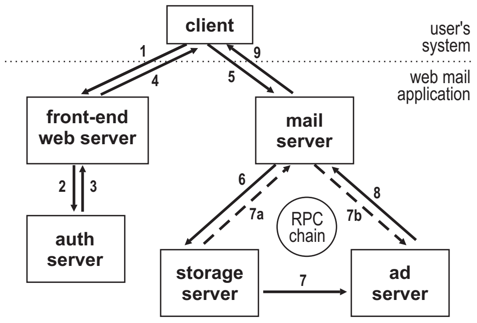

Figure 6: A simplified web mail server that uses RPC chains. The solid line shows the login sequence followed by retrieval of email and ads. The dashed line shows how a system based on standard RPC's would differ. The chain is not used for the web client, since it is outside the system. It is used in the communication between mail, storage, and ad servers.

4.2 Web mail application

Web applications are generally composed of multiple tiers or services: there are front-end web servers, authentication servers, application servers, and storage and database servers. Some of these tiers, namely the web servers and application servers, play the role of orchestrating other tiers, and they tend to keep very little user state of their own, other than soft session state. This is a propitious setting for RPC chains, because performance gains can be realized by optimizing the communication patterns of the various services. We demonstrate this point with a sample application.

We consider a typical web mail application. There are web servers that handle HTTP requests, authentication servers and address-book servers that are shared with other applications, email storage servers that store the users' mail, and ad servers that are responsible for displaying relevant ads. These services can be located in multiple data centers, for several reasons: (1) no single data center can host them all; (2) a service may have been developed in a particular location and so it is hosted close by; (3) for performance reasons, it may be desirable for some services to be located close to their users (e.g., users created in Asia may have their mailbox stored in Asia), though this is not always achievable (e.g., an Asian user travels to the U.S. and his mailbox is still in Asia); and (4) a service may need high availability or the ability to withstand disasters.

We implemented a simple web mail service as shown in Figure 6, to study the benefits of RPC chains in such a setting. Our web mail system consists of a front-end server that authenticates users by verifying their logins and passwords. Upon successful authentication, the front-end server returns a cookie to the client along with the name of an email server. The client then uses the cookie to communicate with the email server to send and receive email messages. Upon receiving a client request, the email server first verifies the cookie, then calls the back-end storage server to fetch the appropriate emails for the user. Finally, the mail server sends the message to an ad server so that relevant ads can be added to the messages before they are returned to the client.

Note that the adding of ads to emails imposes a significant overhead on performance. This is of particular concern because one of the primary performance goals of a webmail service is to minimize the response time observed by clients. In addition, emails and ads cannot be fetched in parallel, since relevant ads cannot be selected without knowing the contents of the emails. It is also difficult to pre-compute the relevant ads because the relevance of ads may change over time.

Using RPC chains, we can mitigate some of the ad-related overheads. Even though we can only fetch ads after fetching the emails, we can eliminate one latency hop from the communication path of the web mail application by creating a chain that causes emails to be sent directly from the storage server to the ad server, without having to go through to email server (as shown in step 7 of Figure 6). Once the ad server has appended the appropriate ads to the emails, the emails can be sent to the email server which then returns it to the client. In Section 5.4, we evaluate the benefit of using RPC chains to improve the communication pattern in this fashion.

5 Evaluation

We now evaluate RPC chains. We start with some microbenchmarks, in which we measure the overhead of chaining functions and we compare RPC chains versus standard RPC's. We then evaluate the storage and web applications to demonstrate the performance improvements provided by RPC chains. The general question we address is when are RPC chains advantageous and what are the exact benefits.

5.1 Setup

In this section, we present the evaluation of our storage and multi-tier web application. Our experimental setup consists of ten machines in four geodistributed sites in a corporate network that spans the globe. We had machines in 4 sites: (1) Mountain View, California, USA, (2) Redmond, Washington, USA, (3) Cambridge, United Kingdom, and (4) Beijing, China. The measured latency and throughput of the links between these sites are shown in Figure 7.

5.2 Microbenchmarks

5.2.1 Overhead of chaining functions

In our first experiment, we evaluate the overhead imposed by chaining functions (pieces of client code) at servers. We considered chaining functions of three sizes, 621 bytes, 5 KB, and 50 KB, corresponding to small, medium, and large functions.

We first measured the time it takes to compile a function at run-time. The results are shown in the first two columns of Figure 8, averaged over 10 runs (± refers to standard error). We used a 3 Ghz Intel Core 2 Duo processor running Windows Vista Enterprise SP1. The functions were written in C# and compiled using Microsoft Visual Studio 2008.

(a)

	Redmond	Beijing	Cambridge
Mt.View	32 ms	180 ms	240 ms
Redmond		146 ms	210 ms
Beijing			354 ms

(b)

	Redmond	Beijing	Cambridge
Mt. View	6.3 MB/s	2.1 MB/s	1.4 MB/s
Redmond		8.5 MB/s	8.6 MB/s
Beijing			2.4 MB/s

Figure 7: (a) Ping round-trip times and (b) bandwidth of TCP connections between pair of sites.

Source size (KB)	Compile time (ms)	Compiled size (KB)
0.6	45.7 ± 0.3	0.4
5	47.1 ± 0.4	4.6
50	76.0 ± 0.3	15.9

Figure 8: Overhead for compiling chaining functions and storing compiled code.

We also did a linear regression with a larger set of points (17 sizes, with 10 runs each) and found that the cost of compilation is 44.8 ms plus 1 ms for each 5000 bytes of source code. We see that there is a large initial compilation cost of tens of milliseconds, which we do not want to pay every time we call the server in a chain.

We measured the size of the compiled code, shown in the third column of Figure 8. We see that it is very small (we initially thought it would be large, but this is not the case). This allows the server to cache even tens of thousands of chaining functions in less than 50 MB, which justifies our choice of doing so.

5.2.2 RPC chain versus standard RPC

In our next experiment, we compare the latency of an RPC chain versus standard RPC. We used the smallest non-trivial chain, which goes through two servers (A

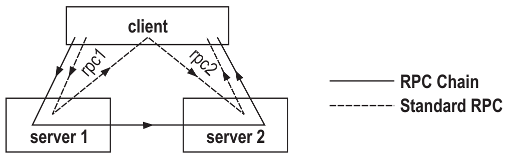

Figure 9: Executions used in the experiment of Section 5.2.2.

chain that goes through only one server is the same as an RPC), and compare it against a pair of consecutive RPC's going to the two servers, as shown in Figure 9. To isolate concerns, the service executed at each server is a no-op.

The figure makes it clear that the RPC chain incurs one fewer hop than the pair of RPC calls. What is not shown is that the RPC chain has potentially two overheads that the pair of RPC calls do not: (1) even if the client needs the response from server 1 but server 2 does not, the data is still relayed through server 2, and (2) the client needs to send state for the chaining function to execute at server 1. The first overhead can be avoided through a simple extension to RPC chains to allow each server in the chain to send some data to the client (Section 7.1).

We now consider the second overhead, and examine the question of how much state the client can send while still allowing the RPC chain to be faster than the pair of RPC calls. We assume that the chaining function is already cached at server 1, which is the common case for frequent chains.

Back-of-the-envelope calculation. We start with a simple calculation. Let S be the size of the state sent by the client for the chaining function at server 1. Then, in terms of total latency, the RPC chain saves one network latency but incurs $S/\mathit{link\_bandwidth}$ to send the state. Thus, the RPC chain fares better as long as $\mathit{link\_latency} > S/\mathit{link\_bandwidth}$, or

$$S < \mathit{link\_latency} \times \mathit{link\_bandwidth}$$

For wide area links, the latency-bandwidth product can easily be in the tens to hundreds of kilobytes or more.

Experiment. We executed the RPC chain and the pair of RPC's. The client was located in Redmond while the servers were in Mountain View. (Because both servers were in the same site, this setup favors the RPC chain by an additional network latency; we later explain the case when the servers are far apart.)

Figure 10 shows the client end-to-end latency as a function of the state size (error bars show standard error). For the standard RPC execution, state size does not affect total latency, since this state simply stays at the client. The total latency was 75±1 ms. For the RPC chain, the latency naturally increases with the state size. The point at which both lines cross is at ≈150 KB. This is a fair amount of state to send in many cases—definitely much more than we needed in either of our applications.

If servers 1 and 2 were far apart, this would shift the RPC chain line up by the corresponding extra latency. For example, if the latency from server 1 to server 2 were 15 ms, the lines would cross at ≈100 KB (assuming the distance from client to server 2 remains the same), which is still a reasonable state size (and much more than we needed in our applications).

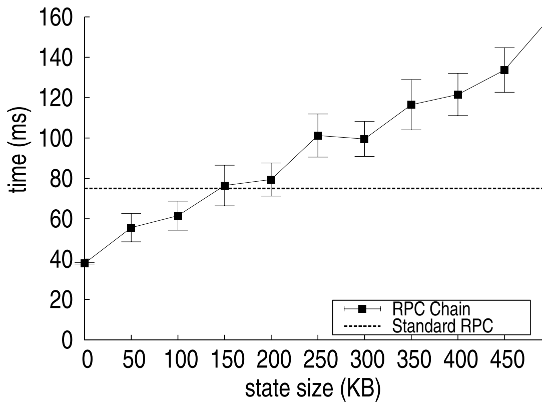

Figure 10: Execution time using an RPC chain versus standard RPC to call 2 servers.

5.3 Storage application

We now evaluate the use of RPC chains for the storage application described in Section 4.1.

5.3.1 Copy performance

In our experiments, we copy data from one storage server to another using two utilities: one that uses RPC chains, called *Chain copy*, and another that uses standard RPC's, called *RPC copy*. Both utilities use pipelining, so that the client has multiple outstanding requests on either server. We also tried using the operating-system provided "copy" program, but it performed much worse than either Chain copy or RPC copy, because it it reads and writes one chunk of data at a time (no pipelining).

In our first experiment, a single client copies a file of variable size (25 KB, 100 KB, 250 KB, and 500 KB) between two servers, and we measure the time it takes. We vary the location of the client (Mt. View, Redmond, Beijing) and fix the location of the servers in Mt. View. In the setting where both the client and the servers were in Mt. View, we placed them in two separate subnets, where the ping latency between the two was 2 ms and TCP bandwidth was 10 MB/s.

Figure 11 shows the results. Each bar represents the median of 40 repetitions of the experiment. As we can see, Chain copy provide considerable benefits in every case, compared to RPC copy. The benefits are greater for larger files and longer distances between client and servers. In a local setting, the copying time is reduced by up to factor of 2, while in the longest-distance setting (Beijing-Mt. View), the reduction is up to a factor of 5.

Another benefit of using Chain copy (not shown) is a reduction by a factor of two in (a) the aggregate network bandwidth consumption, and (b) the client bandwidth consumption. This reduction is important because links

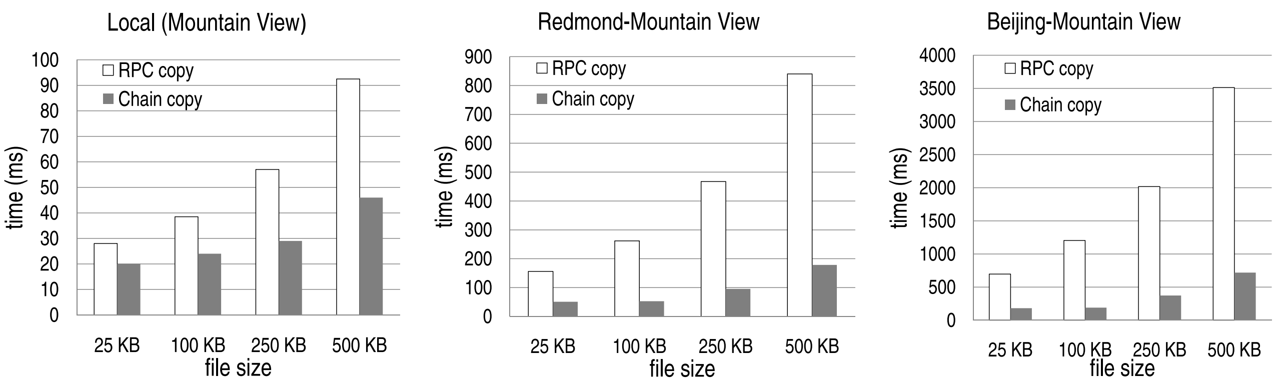

Figure 11: Comparison of RPC copy and Chain copy under various settings. **(Left)** Client and servers are in the same site in Mt. View. **(Center)** Client is in Redmond and servers are in Mt. View. **(Right)** Client is in Beijing and servers are in Mt. View.

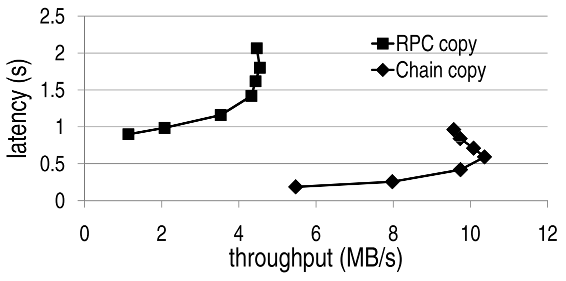

Figure 12: Throughput-latency of RPC copy and Chain copy. Latency is the time to copy a 128 KB file, and throughput is the rate at which files are copied.

connecting data centers have limited bandwidth and/or are priced based on the bandwidth used.

In our next experiment, we vary the number of clients simultaneously copying files from one server to another, and measure the resultant throughput and latency of the system. This allows us to observe the behavior of the system under varying load as well as measure the peak throughput of the system. As before, the client machine was located in Redmond and the servers were located in Mt. View. We ran multiple client instances in parallel on the client machine, each client copying 1000 files in succession, each file measuring 256 KB in size. We measure the time that each client takes to complete copying 1000 files, and compute conservative throughput and latency numbers based on the slowest client.

Figure 12 shows the results of the experiment. For both RPC copy and Chain copy, the average latency decreases as the amount of workload placed on the system increases. Initially, the increase in workload also results in an increase in the aggregate throughput of the system, but once the system becomes saturated, any increase in workload only increases latency without any gain in throughput. Our results show that RPC copy is able to sustain a peak throughput of 4.5 MB/s. This peak throughput occurs when the network link between the client and the servers, which had a bandwidth of 6.3 MB/s, becomes saturated. Since Chain copy does not require that the data blocks of the files being copied actually flow through the client, it was not subject to this limitation and was thus able to achieve a higher peak throughput of 10.4 MB/s. Rather than a network bandwidth limitation, Chain copy's throughput is limited by the servers' ability to keep up with requests.

5.3.2 Benefit of chain composition

In this experiment, we measure the benefit of composing RPC chains. We use two chains: one for copying from one server to another (as above) and the other for primary-backup replication of the second server (as in Figure 5). We compare two systems that use RPC chains; one system uses chain composition to combine the two chains, while the other has composition disabled. In the experiment, one client copies one file of variable size from the non-replicated server to the replicated server. The client is in Cambridge, the source server is in Mt. View, the primary of the destination server is in Mt. View, and the backup of the destination server is in Redmond.

Figure 13 shows the result. As we can see, composing the chain reduces the duration of the copy by 12%-20%, with larger files having a greater reduction. Without composition, the destination server has to handle both requests from the source server as well as the replies from the backup server. Composition reduces the load on the destination server by allowing the backup server to send replies directly to the client. In addition, composition

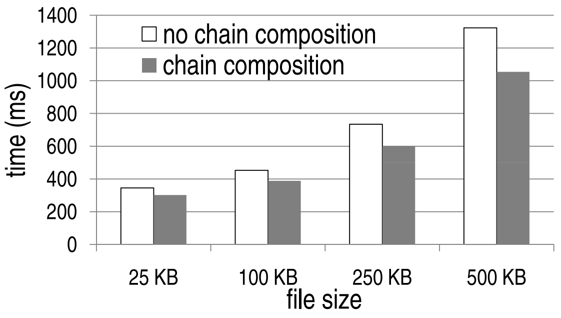

Figure 13: Benefit of chain composition.

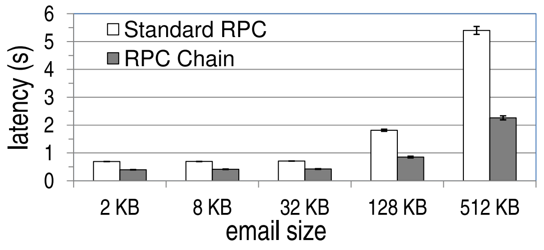

Figure 14: RPC chain in web mail application.

eliminates the unnecessary messages from the backup server to the destination server, reducing the amount of bandwidth consumption. A combination of these factors allow composition to improve the overall performance of the system. As file size increases, the setup cost becomes relatively small compared to the actual cost of executing the chains. This makes the impact of the more efficient chain that resulted from composition more apparent.

5.4 Web mail application

We now describe the evaluation of the web mail application presented in Section 4.2. In our experimental setup, we placed the client in Mountain View, the mail server and the authentication server in Redmond, and all other servers in Beijing. This setup emulates the case where a user from Asia travels to the US and wants to access web mail services that are hosted in Asia. Since the web mail provider may have servers deployed worldwide, the user can be directed to a mail server and an authentication server (Redmond) that is close to his current location (Mountain View). However, user-specific data is stored on servers close to the user's normal location (Beijing), so the mail server has to fetch data from those machines.

Specifically, after receiving a cookie from the client and verifying the client's identity, the mail server must fetch the client's email from the storage server followed by appropriate ads from the ad server, both of which are located in Beijing. A traditional system implemented using RPC's would have the mail server contact the storage server, fetch the user's emails, then contact the ad server to retrieve relevant ads. However, in our setting, where the mail server is located close to the client but far away from the storage server and ad server, traversing the long links between Redmond and Beijing four times would be less than ideal. As described in Section 4.2, RPC chains allow us to eliminate unnecessary network traversals. In this case, our RPC-chain-enabled mail server sends emails directly from the storage server to the ad server before returning the result to the mail server, halving the number of long link traversals.

We measure the client perceived latency of opening an inbox and retrieving one email: the client first contacts the front end authentication server to authenticate herself, then she sends a read request to the mail server to retrieve a single email. We measure the time it takes for the client to receive the email, which is appended with an ad whose size is small relative to the size of the email. We vary the size of the email that is fetched, and for each size, we repeated the experiment 20 times.

As shown in Figure 14, RPC chains consistently reduces the client perceived latency of the web mail application. As the size of the email increases, the latency improvement from using RPC changes also increases. Overall, we found that the use of RPC chains reduced the latency of the web mail application by 40% to 58% when compared to standard RPC's.

We note that the significant performance gains of using RPC chains comes at a very low cost of implementation. For the web mail application, the effort involved in enabling RPC chains was mainly in terms of implementing chaining functions which totaled a mere 48 lines of C# code. In general, a simple way for existing applications to benefit from RPC chains is to identify the critical causal path of RPC requests, and replace that path with an RPC chain. The effort is that of writing a single RPC chain; in the worst case, one can do it from scratch. The harder problem is finding the critical causal path, which has been an active area of research (e.g., [1]).

6 Limitations

We now describe some limitations of RPC chains.

Chaining state cannot always be sent. RPC chains are not appropriate if the chaining state is large or if it cannot be determined when the client starts the chain. For example, suppose that (1) A calls B using an RPC, (2) A gets a reply, and (3) depending on the state of a sensor or some immediate measurement at A, A then calls C or D. It is not possible to use an RPC chain $A \rightarrow B \rightarrow (C \text{ or } D)$, because the choice of going to C versus D must be made at A where the sensor is.

Programming with continuations. To use RPC chains, developers need to make use of continuation-style programming. This can be much harder than programming using sequential code, because continuations must explicitly keep track of all their state. Continuations are notoriously hard to debug, because there is no simple way to track the execution that led to a given state.

We note, however, that programming with continuations is already tolerated in code that uses asynchronous RPC's and callbacks. Moreover, one could perhaps write a tool that automatically produces continuations from sequential code, using techniques from the compiler literature (see, e.g., [3]).

Terminating chains. When an application terminates, it is usually desirable to release its resources and halt all its activities. However, if the application has outstanding RPC chains, it is not easy to terminate them. This problem exists with traditional RPC's as well (there is no easy way to terminate a remote procedure), but it is worse with RPC chains because the remote servers involved may not be known.

RPC chains are designed for relatively short-lived executions, and for these uses, this problem is less of a concern, because a chain soon terminates anyways. The only exception is a buggy chain that runs forever. For such chains, the RPCC library can impose a maximum chain length, say 2000 hops, and throw an exception after that.

7 Extensions

We now discuss some extensions of RPC chains.

7.1 Intermediate chain results

If a client wants to receive some results from intermediate servers of the chain, these results need to be relayed through the chain. If the amount of data is large, it can impose a significant overhead. We can extend RPC chains to address this issue, by allowing each server in the chain to directly return some data to the client. This data is application-specific and is returned by the chaining function. Thus, we add a new field, `client-response`, to the `nexthop` result of a chaining function. The RPCC library sends `client-response` to the client concurrently with continuing the chain.

What happens under chain composition? In this case, the "client" that gets `client-response` is the server that created the sub-chain. The name of these creators, at each level of composed chain, are kept in the chain stack (the chain stack is explained in Section 3.4).

7.2 Dealing with large chaining states

The chaining state is the state that the client sends along the chain to execute the chaining functions. If this state is large, this can incur a significant state overhead. Two optimizations are possible to mitigate this cost.

Fall-back to standard RPC. As explained in Section 3.7, we can execute a chain in interactive mode, which causes the chain to go back to the client at every step. This is effectively a fall-back to standard RPC, causing all chaining functions to execute at the client, which eliminates the overhead of sending the chaining state, at the cost of extra network delays. We explored this trade-off in Section 5.2.2. It is possible to have the RPCC library gauge the size of the chaining state before starting the chain, and if the state is larger than some threshold, execute the chain in interactive mode. The threshold can be chosen dynamically based on previous executions of the same chain, in an adaptive manner. By doing so, an RPC chain will always perform at least as well as standard RPC's, modulo the small computational overhead of executing chaining functions and the time it takes to adapt. However, in the applications we examined in this paper, we did not need this technique because the chaining state was always small.

Hiding latency. In our implementation, servers wait to receive the chaining state before executing the next service function in the chain. This waiting is not necessary, because the service function depends only on its parameters, not on the chaining state (the chaining state is only needed for the chaining function, which executes later). Therefore, a natural optimization is to start the service function even as the chaining state is being received. If the service function takes significant time to complete, (e.g., it involves disk I/O or some lengthy computation), this will mask part or all of the latency of transmitting the chaining state.

7.3 Chaining proxy

As we said, chaining functions are *portable code* that do not have to execute at the server. They can execute at a designated *chaining proxy* machine, to avoid any overhead at the server. Doing so incurs extra communication, but if the chaining proxy is geographically close to the server, this cost is small relative to that of a wide-area hop. To choose the chaining proxy, we can use a simple mapping from servers to nearby proxies configured by an administrator.

8 Related work

RPC chains utilize two well-understood ideas in the context of remote execution: *function shipping*, and *continuations*.

Function shipping is the general technique of sending computation to the data rather than bringing the data to the computation. It is used in some systems where the cost of moving data is large compared to the cost of moving computation. For example, Diamond [10] is a storage architecture in which applications download searchlet code to disk to perform efficient filtering of large data sets locally, thereby improving efficiency. RPC chains use function shipping to send chaining logic.

A continuation [17] refers to the shifting of program control and transfer of current state from one part of a program to another. Extending this to *distributed continuations* is a natural step, allowing a continuation to shift program control from one processor to another. Several works in the parallel programming community give high-level programming continuation constructs and specify their behavior formally, e.g., [12, 11]. Distributed continuations were exploited to enhance the functionality of web servers and overcome the stateless nature of HTTP interaction. By comparison, the RPC chain is a generic mechanism that is independent of the service provided by servers. RPC chains support complex chaining structures, and can be used with a diverse set of servers.

The above mentioned ideas for code mobility, and others, are leveraged in a variety of high-level programming paradigms for distributed execution. Distributed workflows, e.g., [5, 22], can use distributed continuations to distribute a workflow description in a decentralized fashion. MapReduce [6], and Dryad [23] are programming models for data-parallel jobs, such as a data mining calculations, which process large amounts of data in batches. These systems target self-contained jobs that execute for substantial periods, while RPC Chains are intended for short-lived remote executions in an environment with many diverse services that are possibly developed independently of their applications. Mobile agents have been extensively studied in the literature and many systems have been built, including Telescript/Odyssey [19], Aglets [4], D'Agents [8], and others (see e.g., [20, 9]). A mobile agent is a process that can autonomously migrate itself from host to host as it executes; migration involves moving the process's current state to the new host and resuming execution. The motivation for mobile agents include (a) bringing processes closer to the resources they need in a given stage of the computation, and (b) allowing clients to disconnect from the network while an agent executes on their behalf. An RPC Chain can be considered as a mobile agent whose purpose is to execute a series of RPC calls. However, mobile agents are much more general and ambitious than RPC Chains (which possibly contributed to their eventual demise): they have social abilities, being able to adjust their behavior according to the host in which they are currently executing; they can learn about execution environments never envisioned by their creators; and they can persist if the clients that created them disappear. Much of the literature regarding mobile agents is about security (how agents can survive malicious hosts, and how hosts can protect themselves against malicious agents) and language support for code mobility (how to write programs that can transparently move to other machines). For RPC chains, security is a smaller concern in the trusted data center and enterprise environments that we consider, and we are not concerned about transparent mobility.

Some related work includes more targeted uses of mobile code. Work on Active Networks introduced network packets called *capsules*, which carry code that network switches execute to route the packet (see [18] for a survey). This provides a general scheme for extending network protocols beyond the existing deployed base, and allows for more dynamic routing schemes. In contrast, RPC chains are aimed at higher-level applications, and their main purpose is to eliminate communication hops when a client needs to call many services in succession.

Distributed Hash Tables (e.g., Chord [16], CAN [14], Pastry [15], Tapestry [24]) have a *lookup* protocol, for finding the host responsible for a given key. Such protocols generally need to contact several hosts successively, and this can be done in two ways. In an *interactive* lookup protocol, the host that initiates the lookup operation issues RPC's to each host in succession. A *recursive* lookup protocol [7] works like a routing protocol: the host that initiates the operation contacts the first host in the sequence, which in turn contacts the next one, and so forth; when a host finds the key, it contacts the request initiator directly. This protocol is hard-coded into the lookup operation, and it is executed by a set of servers that implement this operation. In contrast, RPC chains provide a generic chaining mechanism that is independent of the operation (service function) executed.

Finally, SOAP [21] is a protocol that supports RPC's using XML over HTTP. It has the notion of intermediaries that can process a SOAP message (RPC) before it reaches the final recipient. However, there is no client logic that routes and transform messages, and the notion of a pre-specified distinguished final recipient is inherent to SOAP. Typical uses for intermediary nodes include blocking messages (firewall), buffering and batching of messages, tracing, and encrypting/decrypting messages as it passes through an untrusted domain.

9 Conclusion

We proposed the RPC chain, a simple but powerful primitive that combines multiple RPC invocations into a chain, in order to optimize the communication pattern of applications that use many composite services, possibly developed independently of each other. With RPC chains, client can save network hops, resulting in considerably smaller end-to-end latencies in a geodistributed setting. Clients can also save bandwidth because they are not forced to receive data they do not need. We demonstrated the use of RPC chains for a storage and a web application, and we think RPC chains could have many more applications beyond those.

Acknowledgements. We thank Rama Subramanian, our shepherd Michael J. Freedman, and the anonymous reviewers for helpful comments. This research was supported in part by NSF Grant 0424422 and U.S. Dept. of Homeland Security Grant 2006-CS-001-000001-02.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 74–89.

[2] AMAZON.COM, INC. Amazon simple storage service: Copy proposal. `http://doc.s3.amazonaws.com/proposals/copy.html`.

[3] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] ARIDOR, Y., AND OSHIMA, M. Infrastructure for mobile agents: Requirements and design. In *Workshop on Mobile Agents* (Sept. 1998), pp. 38–49.

[5] BARBARÁ, D., MEHROTRA, S., AND RUSINKIEWICZ, M. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management, Special Issues on Multidatabases 7*, 1 (Winter 1996), 5–15.

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *ACM Symposium on Operating System Design and Implementation* (Dec. 2004), pp. 137–150.

[7] FREEDMAN, M. J., LAKSHMINARAYANAN, K., RHEA, S., AND STOICA, I. Non-transitive connectivity and DHTs. In *Conference on Real, Large Distributed Systems* (Dec. 2005), pp. 55–60.

[8] GRAY, R., KOTZ, D., NOG, S., RUS, D., AND CYBENKO, G. Mobile agents: The next generation in distributed computing. In *Aizu International Symposium on Parallel Algorithms and Architectures Synthesis* (Mar. 1997), pp. 8–24.

[9] HARRISON, C. G., CHESS, D. M., AND KERSHENBAUM, A. Mobile Agents: Are they a good idea? In *International Workshop on Mobile Object Systems* (July 1996), pp. 25–47.

[10] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *USENIX Conference on File and Storage Technologies* (Mar. 2004), pp. 73–86.

[11] JAGANNATHAN, S. Continuation-based transformations for coordination languages. *Theoretical Computer Science 240*, 1 (June 2000), 117–146.

[12] MOREAU, L. The PCKS-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming* (Apr. 1994), pp. 424–438.

[13] NIELSEN, J. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Indianapolis, 1999.

[14] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R. M., AND SHENKER, S. A scalable content-addressable network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Aug. 2001), pp. 161–172.

[15] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms* (Nov. 2001), pp. 329–350.

[16] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Aug. 2001), pp. 149–160.

[17] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation 13*, 1-2 (Apr. 2000), 135–152.

[18] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A survey of active network research. *IEEE Communications Magazine 35*, 1 (Jan. 1997), 80–86.

[19] WHITE, J. Telescript technology: The foundation for the electronic marketplace, 1994. Unpublished manuscript. White paper, General Magic, Inc.

[20] WHITE, J. Mobile agents white paper, 1996. Unpublished manuscript. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.7931`.

[21] WORLD WIDE WEB CONSORTIUM. SOAP version 1.2. `http://www.w3.org`.

[22] YU, W., AND YANG, J. Continuation-passing enactment of distributed recoverable workflows. In *ACM Symposium on Applied Computing* (Mar. 2007), pp. 475–481.

[23] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *ACM Symposium on Operating System Design and Implementation* (Dec. 2008), pp. 1–14.

[24] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004), 41–53.