Quatrain: Accelerating Data Aggregation between Multiple Layers

Jinglei Ren, Yongwei Wu, *Member, IEEE*, Meiqi Zhu, and Weimin Zheng, *Member, IEEE*

Abstract—Composition of multiple layers (or components/services) has been a dominant practice in building distributed systems, meanwhile aggregation has become a typical pattern of data flows nowadays. However, the efficiency of data aggregation is usually impaired by multiple layers due to amplified delay. Current solutions based on data/execution flow optimization mostly counteract flexibility, reusability, and isolation of layers abstraction. Otherwise, programmers have to do much error-prone manual programming to optimize communication, and it is complicated in a multithreaded environment. To resolve the dilemma, we propose a new style of inter-process communication that not only optimizes data aggregation but also retains the advantages of layered (or component-based/service-oriented) architecture. Our approach relaxes the traditional definition of procedure and allows a procedure to return multiple times. Specifically, we implement an extended remote procedure calling framework Quatrain to support the new multireturn paradigm. In this paper, we establish the importance of multiple returns, introduce our very simple semantics, and present a new synchronization protocol that frees programmers from multireturn-related thread coordination. Several practical applications are constructed with Quatrain, and the evaluation shows an average of 56% reduction of response time, compared with the traditional calling paradigm, in realistic environments.

Index Terms—Distributed programming, distributed systems, data communications, interfaces, system integration and implementation, asynchronous/synchronous operation, client/server, frameworks, patterns, procedures, functions, and subroutines

1 Introduction

Complexity of building large distributed systems is controlled via `layers` abstraction. We use the term layer to mean a tier in the multi-tier architecture [1], a component in the component-based architecture [2], or a service in the service-oriented architecture (SOA) [3]. Under such abstraction, a system is divided into layers and layers interact with each other via defined interfaces. Layers raise the isolation, flexibility, and reusability of elements in a distributed system and therefore largely facilitate system development.

Between layers, *remote calling* acts as a pervasive and fundamental programming paradigm. Specific forms include: the traditional remote procedure call (RPC), such as Java remote method invocation (RMI) [4], Microsoft. NET remoting [5] and Apache Thrift [6]; SOAP web services [7]; RESTful HTTP request and response [8] where the callee is a resource; etc.

Meanwhile, data aggregation becomes a typical pattern among and inside data centers. For example, on the shuffle phase of MapReduce [9], a reducer gathers its portions of intermediate data from all mappers. Actually, the partition/aggregate design pattern constitutes foundation of many large scale applications [10], such as web search, and social network content composition. Under the application layer, socket-level logs also identify the "scatter-gather" (a synonym of aggregation here) traffic pattern [11] in data centers.

However, when data aggregation meets multiple layers, traditional remote calling mechanisms are caught in a dilemma. As illustrated in Fig. 1, when a client making a remote call to the server interface for data aggregation, not all required data are produced at the same time during either sequential (a) or parallel (c) execution. Some data may get gathered much faster than others, but the server does not return any data until the slowest one is ready, since the called procedure can only return once. The meaningless wait for slow data results in increased response time (the average arrival time of required data).

In this paper we propose Quatrain, a new programming and communication mechanism that solves the problem. Quatrain challenges the traditional assumption that a procedure returns only once, and allows the called procedure to make multiple returns during its execution. We extends a most popular form of remote calling, RPC, to implement our prototype. Application of RPC can be seen in MapReduce [9], RAMCloud [12], the Facebook infrastructure [6], etc.

Quatrain allows the server-side procedure to return partial replies at *any* position for *any* times within its main thread or in *any* other thread it creates[1]. Consequently, the caller receives partial replies as soon as they are ready, without waiting for the whole to return for once, as illustrated in Fig. 1(b) and (d). Quatrain enables the caller to process parts of data in parallel to form pipelines, and also overlaps network transmission with data production. Moreover, granularity of return (e.g., each record of data, or a specific subset of records) can be flexibly controlled by the procedure to reach optimal performance.

1. For convenience, we express parallelism by "thread" in this paper.

• *The authors are with the Department of Computer Science and Technology, Tsinghua National Laboratory for Information Science and Technology (TNLIST), Tsinghua University, Beijing 100084, China, and Research Institute of Tsinghua Universityin Shenzhen, Shenzhen 518057, China. E-mail: wuyw@tsinghua.edu.cn.*

Manuscript received 27 Sep. 2012; revised 08 Dec. 2012; accepted 12 Dec. 2012; published online 23 Dec. 2012; date of current version 29 Apr. 2014. Recommended for acceptance by K. Li.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2012.298

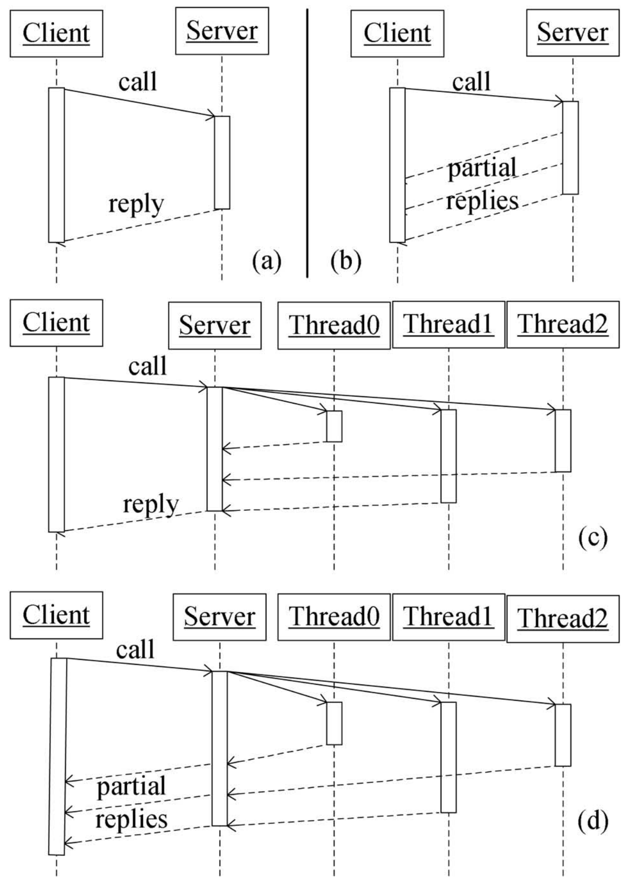


Fig. 1. Execution flows: (a) traditional RPC with sequential execution; (b) multi-return RPC with sequential execution; (c) traditional RPC with parallel execution; (d) multi-return RPC with parallel execution.

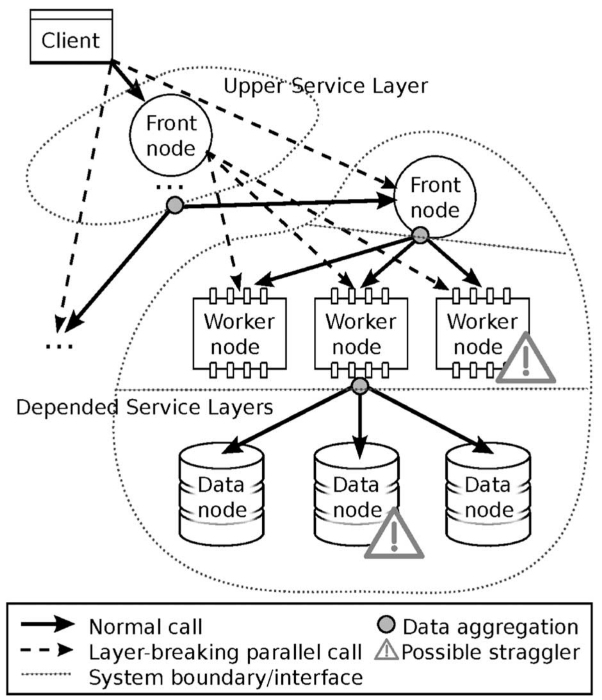


Fig. 2. A typical layered architecture. Solid arrow lines and dashed ones are mutually exclusive, respectively denoting the normal calling and the layer-breaking calling. Gray circles are where data delay may happen.

1.1 Motivation

Delay by layers is practically a serious problem. First, the diversity of arrival times of aggregated data is large, and more diversity causes more delay of early data. According to our evaluation, the diversity can even reach 2 orders of magnitude for a production service (Section 5.3). What is worse, *stragglers* may add much to such diversity. A straggler refers to any node who encounters very low efficiency. Several investigations [13], [14] have revealed that stragglers significantly prolong job completion. Causes of stragglers are quite common, including hardware and software failures, uneven work partition, temporarily overloading, network congestion [11], etc. Second, in a typical layered architecture, the delay accumulates each time data flow across a layer boundary, so that the negative impact is amplified by multiple layers. Third, even minor improvements on response time are significant for service providers. Experiments at Amazon showed 1% sales decrease for every additional 100 ms response time, and experiments at Google showed that increasing the time to display search results by 500 ms reduced revenues by 20% [15]. An Internap white paper [16] describes latency as the "Achilles heel of cloud computing".

Why parallel calling cannot solve the problem? Parallel RPC [17], [18] enables the client to send parallel requests to many servers. Intuitively, making parallel RPC can get early data without delay. However, the current system situations are different with twenty years ago when parallel RPC was invented. Firstly, parallel RPC requires the client to maintain all target servers' information and manually split a single RPC into several ones, but in current practice it is usually hard for clients to decide how work is partitioned or which nodes to call. For example, the work may be partitioned and allocated dynamically to variable nodes by a coordinator according to some schedule policy on the server side. Secondly, a parallel RPC requires the client to manually handle stragglers (slow replies). In large-scale systems, stragglers are notorious and not negligible. Dealing with stragglers adds extra complexity and burden to programmers. Thirdly, the parallel RPC violates layers abstraction. Take a typical service structure in Fig. 2 for example. If the client uses parallel RPC to realize one logic call to a front node, it has to directly call the depended layers (dashed arrow lines from "Client", instead of the solid arrow line); if a component uses parallel RPC to make a logic call to some interface of another layer, it has to directly call the internal components inside that layer in parallel (dashed arrow lines from "Front node"). In either case, the interface or system boundary is broken, violating basic isolation in large systems. Exposure of internal components behind interface or system boundary shall almost definitely cause architectural or security issues. Moreover, in SOA, a depended service may be from a third party and out of control, which makes cross-layer parallel RPC impossible. By contrast, Quatrain still uses a single RPC for one logic request, similar to a traditional one, but covers all the issues internally.

A research trend in next-generation data center networks illuminates another potential advantage of multi-return paradigm. D^3 [19] is a new control protocol that strives to meet flow deadlines by explicit rate control. When flows with the same deadline cannot be all satisfied, D^3 sheds some load to ensure that a good *fraction* of flows meet the deadline, rather than to delay all. However, if the upper-layer application still needs to wait the shed flows, the benefits of such networks would be in vain to end users. This property on the network layer takes better effect when the multi-return paradigm enables the application layer to make use of the partial early data instead of waiting for all.

Quatrain aims at providing a convenient and efficient abstraction to alleviate the delay-by-layers problem, avoiding complex combination of traditional techniques. A similar case is MapReduce [9], which does not aim at solving a problem unsolvable via existing techniques, but provides a neat programming model that frees programmers from involved issues and labor. For a large system, the remote calling primitive, as a most basic code block, should be as small and simple as possible. Without Quatrain, however, a programmer needs multiple parallel/sequential calling *plus* streaming/message-based communication *plus* asynchronous invocation *plus* manually coded thread coordination to realize similar optimization.

1.2 Contributions

We summarize the contributions of this paper as follows. First, we establish the significance of multi-return paradigm in distributed systems. It accelerates data aggregation by reducing average response time, pipelines workload in fine grain by breaking data into smaller blocks, tolerates stragglers by not letting them drag others, overlaps data production with transmission, and well supports some potential next-generation data center networks. More importantly, all these features are achieved in the *precondition* that benefits of layers abstraction are not traded off. Existing solutions hardly hold such advantage.

Second, we introduce new semantics of *preturn* and *reply set*. To keep Quatrain's interfaces as simple as possible is a challenge, and we best limit the changes brought to current languages and conventions. Only one new primitive `preturn` (partial return) is necessary, and the original reply (a number of records) is extended into a `reply set`, whose elements are records of the original reply. The *preturn* requires no extra parameters other than the returned data, and programmers do not have to deal with any multi-return-related threading details.

Third, we design a partially ordering protocol (*POP*) to coordinate all `preturns` and the final signal that notifies the caller to stop waiting. Supporting multi-threading as well as the event-based model adds to the complexity, and these issues should be transparent to programmers. Our protocol guarantees that the end notification is after all `preturns` have sent back their replies, and sufficiently saves programmers' error-prone manual work.

Our work is an improvement to RPC for distributed systems. Although the high-level concept (multi-return) has appeared in some programming languages [20]–[22], using it in distributed environment is never seen. And the difference between them is fundamental, like between normal procedure calling and RPC.

Compared with some data flow optimizing solutions [23], [24], Quatrain provides a more flexible and less intrusive primitive for distributed systems development. Our model is flexible in the sense that it only requires replacement of traditional RPC and is not bound to a specific computing model. Quatrain is not intrusive in the sense that rewriting is only needed to replace original RPC, without any destruction to existing architecture (all logic except RPC-related). In contrast, other frameworks mostly need a thorough switch for existing solutions to a new defined processing model.

Section 2 describes the basic usage of multi-return RPC. Section 3 articulates implementation issues, including the basic architecture, the ordering protocol and scalability. The general application of Quatrain is demonstrated in Section 4, and its effectiveness is evaluated in Section 5. We discuss related work in Section 6, and finally conclude in Section 7.

2 Semantic Design

One critical principle of Quatrain's design is **simplicity**, which is the original intention and the most advantage of RPC. Programmers should still use similar semantics as traditional RPC, and keep a traditional view of the programming logic. Our solution merely introduces two new library-level extensions visible to programmers:

1. `preturn` (partial return) is a server-side library function that sends partial results back to their caller. This function can be called at any position for any times in any created thread. Such flexibility allows fine control on the granularity of each `preturn`. Meanwhile, it requires no extra parameters other than the exact replied object, like `return`.
2. `Reply set` is an client-side object for the caller to enumerate replies, either sequentially or in parallel, until `null` is encountered or `hasMore()` returns false. For each `nextElement()`, the caller is blocked until either a new reply or `null` returns.

2.1 Sample Codes

Based on the simple semantics, programmers can invoke multi-return RPC in a very familiar way. Since the client's processing on replies is application-specific and therefore not our focus, only sample codes for sequential access are posted here (using Java's Enumeration interface in our implementation).

Sample Code 1. Client

```
ReplySet replies = client.invoke(String.
  class, "SampleProcedure", parameter);
//incrementally handle parts
while (replies.hasMore()) {
  // do work on each
  doWork(replies.nextElement());
}
//judge whether all replies arrive
if (replies.isPartial())
  Log.info("Some replies omitted.");
```

A remote procedure can be called via the `invoke()` method as Line 1, or via a proxy instance just like invoking a local method (a compiler can be provided to generate necessary codes for programmers as Thrift [6] does, or even Quatrain can be integrated into Thrift, but these are pure engineering work and not the focus of this paper). The replies come into the ReplySet object in batches, and an enumeration interface offers each element (Line 5). ReplySet's method `hasMore()` may be blocked (Line 3) until next element or end notification arrives, and `isPartial()` tells whether current set covers all expected replies.

The following code snippet shows a multi-threaded user-defined remote procedure on the server side.

Sample Code 2. Server

```
1  public void SampleProcedure(int count) {
2     for (int i = 0; i < count; ++i) {
3        new Thread(new Runnable() {
4           public void run() {
5              preturn("in any thread.");
6              preturn("for many times.");
7           }
8        }). start();
9     }
10    preturn("at any position.");
11 }
```

This procedure creates a number of worker threads (Line 3), and each thread uses preturns to reply to the caller. The 3 `preturns` demonstrate their flexibility (it can be used in any thread for any times at any position). The granularity of `preturn` can be adjusted according to specific applications. Moreover, the `preturn` requires no extra parameters other than the returned data. Other implicit parameters (e.g. the caller to return data to) are all handled within Quatrain for programmers' convenience.

Without Quatrain, programmers have to write several tens of times the number of code lines of the above samples to reach similar optimized functionality. Generally speaking, our work is designed for service/system developers (experts rather than end users) who meet data aggregation and want to wrap all delay-by-layers issues within a simple RPC interface.

3 Implementation

In this section, we present our Java implementation of multi-return RPC based on TCP sockets and Hprose [25] encoder/decoder. After a brief introduction to Quatrain components, we articulate the new synchronization protocol for partially ordering. The issue of scalability is also discussed.

3.1 Architecture

The basic architecture of Quatrain is similar to any traditional RPC implementation [26], except that a management component for reply sets and a thread coordination mechanism for multiple `preturns` are respectively working on the client and the server. Fig. 3 depicts Quatrain's high-level architecture, and features are explained as below.

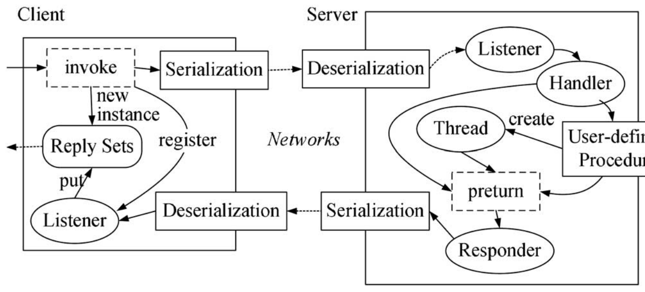


Fig. 3. Framework of multi-return RPC.

3.1.1 Client

Quatrain offers the `invoke()` method for programmers to call remote procedures. Within the method, new connections are established to the server and RPC requests are sent. Then each connection is registered to a *listener*, which *select*s sockets to detect replies from the server. All connections are initialized from the client and are active until the end notification arrives.

On invocation, a new data object `ReplySet` is created. Every reply set is restored on a hash table with a unique *call ID*. Reply sets in such hash table are all waiting for replies. The call IDs go with all requests and replies to have them properly dispatched. The `listener` takes charge in adding all replies to their corresponding reply sets.

3.1.2 Server

On the server side, the daemon thread *listener* utilizes a *selector* to accept connections from clients and reads in requests. Then the `listener` creates a *handler* for each request and puts it on to a execution queue. A thread pool takes charge of executing all queued `handlers`. When it runs, each `handler` invokes the target user-defined procedure via Java reflection. Each `handler` and all its forked threads share a variable that stores the caller's ID.

`preturn` is a method defined in the server and used by any user-defined procedure to send replies. In runtime, it creates a *responder* who does the actual socket write operations. Similar with the `handler`, each `responder` waits for a executor from a thread pool. When it runs, the `responder` serializes the call ID as well as replies, and transmit data through the connection.

Another core utility defined in server is the *Thread* class that extends standard java.lang.Thread. Programmers have to use this `Thread` rather than its origin, because extra operations are necessary to implement the synchronization protocol in Section 3.2. In addition, using a thread pool within a user-defined procedure is also supported. The only requirement is that programmers should create an object of our *Runnable* class (usage is similar with `Thread`). Then it can be queued to the thread pool as normal.

3.2 Partially Ordering Protocol

One critical issue of multiple `preturns` is how to notify the caller at the right time to stop waiting for additional replies.

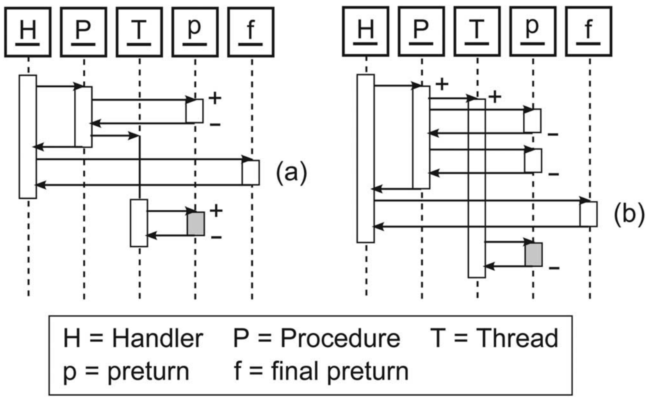


Fig. 4. Counter-examples for naive solutions. Timelines are top-down, and preturns are set on separate timelines for clarity. (a) and (b) denote wrong final preturns when the count becomes zero. Gray blocks are omitted data.

Without such end notification, the client has no way to judge whether all expected replies are returned.

3.2.1 Problem Statement and Complexity

Our goal is to hide all multi-return-related threading details from programmers and introduce no additional semantics other than the simple and flexible `preturn`. We artificially add one final `preturn` in the `handler` to send end notification after invocation of the called procedure. Then if we consider all the network output operations triggered by `preturns` as a set, the set should be in partial order under the relation that any other network output operation is no later than the end notification. Thus we name our new protocol the partially ordering protocol (POP).

Complexity of the problem is demonstrated through some flawed naive solutions, which also vindicate the necessity of a formally verified protocol.

First of all, this problem is not covered by native thread synchronization (like `join()`), because use of threads is arbitrary in the user-defined procedure, and there is a queue between the `handler` and the `responder`. Even if the `handler` and all its forked subsidiary threads have terminated, it is still uncertain when and in what order the actual network outputs happen.

The most intuitive method is to set up for each procedure an atomic reference count that is increased when a `preturn` is invoked and decreased after its network output finishes. Then the final `preturn` waits until the count equals zero. Since we do not determine when a thread actually executes after it starts, a counter-example is shown in Fig. 4(a), where the final `preturn` goes before some later scheduled thread that owns `preturn`.

Despite of the uncertainty of multi-threading, one thing we can assure is when a new thread is created. So there is an improvement that moves increment to thread start. However, Fig. 4(b) shows again a counter-example, where an extra `preturn` called by the procedure decreases the reference count to zero before the thread invokes `preturn`.

Here we have named a few examples among all the pains we took to only find a small possibility of failure after thousands or even tens of thousands of normal calls, which shows the necessity of a formally verified protocol.

3.2.2 Protocol and Verification

POP employs one reference count to realize partially ordering. A more straightforward way of our protocol can use multiple reference counts for different threads, but it is lengthy and less cost-efficient. Our solution requires a formal verification of the validity to use only one consolidated reference count, but makes implementation very concise.

We firstly offer an intuitional description here. The POP's purpose is to arrange, within a complex task, a final action after others (recall that pure thread synchronization is not enough). Then we find that workers of a task finish their work in stages. The protocol guarantees that each stage of each worker makes a mark before its next stage is triggered, and the triggered stage revokes that mark after it finishes. So the final action should wait until all marks are revoked. These marks can be consolidated to a single reference count, and we mapped the relations and actions to a graph for formal verification.

A reference count is an integer variable assigned to a called procedure and visible to all forked threads. It supports two *atomic* operations: P decreases the count by one, and V increases the count by one. Since its initial value is zero and should finally reach zero again before the end notification, we can make all P and V operations into pairs (see proof in Appendix Property 1). For convenience, we color all P/V pairs and denote a pair with the kth different color by $P(k)/V(k)$, where $k \in \mathbb{N}^+$. For the kth color, we set an independent sub-count r_k (only for verification purpose), and suppose $P(k)/V(k)$ performs on r_k. Then we have $R(k) = \sum_{i=0}^{k} r_i$, so the actual reference count equals to R(t) where t is the max color number.

If one thread creates another, we call it the *parent* of the created one which is then called the *child*. All threads derive from a *root* thread (in our case, the called procedure can be seen as the root). Each type of threads is associated with a different color.

Then we use a directed acyclic graph to express the relationship between threads and determine which sub-count a P/V operation should perform on. The graph models the relationship between static thread types (or classes) including dynamic type binding, rather than the runtime thread instances. We made such choice because it facilities the adoption of the protocol, and is friendly to use by compilers or program analysis tools.

In the graph $G = (V, E)$, each thread type is denoted as a vertex $v \in V$, and all *event*s required to be partially ordered are abstracted to one vertex $t \in V$, namely both threads and events are separately expressed as vertexes. An edge $(u, v) \in E$ lies from a parent thread u to its child v if and only if $u \neq v$. If a thread u triggers an event one or more times, there exists $(u, t) \in E$. Another vertex we distinguish is the root thread s. Then we can view the graph as a flow network from the source s to the sink t. The color of a vertex v is numbered as the length of the longest path from s to v. Fig. 5 illustrates an example thread structure.

Now we map the P/V operations onto the graph:

Map 1. Each vertex v is associated with a P operation with the vth color if and only if t is reachable from v.

Map 2. Each edge (u, v) corresponds to a V operation with the vth color if and only if v is associated with a P operation.

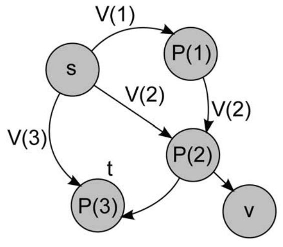


Fig. 5. An example of a thread structure.

Considering events t as a special thread type, the protocol is as follows:

Rule 1. When a thread of type τ_1 creates a thread of type τ_2, the V operation associated with the edge from τ_1's vertex to τ_2's vertex is carried out if it exits.

Rule 2. When a thread of type τ finishes, the P operation associated with the τ's vertex is carried out if it exists.

Rule 3. The final event is scheduled after all V-operations of root.

Rule 4. The final event does not happen until the reference count equals to zero.

Before the main proof of POP, we have to demonstrate its two important properties.

Property 1. *P and V operations of the same color are in pairs.*

Proof. According to Map 2, all edges pointing to a P operation are associated with V operations. Meanwhile, considering events as a special type of threads, we have the reasoning as follows.

If any P operation is on a thread, Rule 1 ensures that the thread is created with a V operation (which has the same color). On the other hand, the thread must end at some time, so its associated P operation must be invoked according to Rule 2.

Therefore, all P and V operations with the same color are in pairs, namely Property 1 holds. □

Property 2. *Any $V(k)$ is invoked from the root thread s, or between a first V(m) and a second P(m), where $m < k, k \geq 2$.*

Proof. We only need to consider the $V(k)$s invoked by non-root threads.

First, we are to verify that any $V(k)$ is located on an edge (u, v), where u is associated with $P(m)$ and $m < k$. According to Map 1, such edge must start from a vertex that has some P operation. Then we suppose $m \geq k$, the longest path from s to v passing u is at least of length $m + 1 > k$, which conflicts with the definition of v's color number. So, we must have $m < k$.

Then, with the same reasons in the verification of Property 1, the vertex u must be wrapped in a first V operation and a second P operation. As the $V(k)$ must be invoked while the u's thread is still alive, it must be invoked within that pair.

Therefore, the V(k) must be invoked between a first $V(m)$ and a second $P(m)$ where $m < k$. Then, Property 2 holds. □

Based on these rules and properties, we guarantee that the final event is ordered after all other events. Below goes our verification.

Proof. According to Rule 4, to proof that the final event happens after all other events, we only have to demonstrate that, at the point of final event, the zero reference count indicates end of all other events.

We carry out a mathematical induction to reach the conclusion. First of all, the color number is a finite natural number, and we suppose its max value is t, since it equals to the color number of vertex t. For convenience, we denote s with color number 0, and consider execution of events t as $V(t + 1)$.

Statement If $R(n)$ becomes zero (not the initial state), all $V(n + 1)$ operations must have been invoked.

Base As for $n = 0$, all V(1) edges come from root, according to Property 2. Then Rule 3 guarantees that, at the point of final event, all these V(1) operations have executed.

Induction As for $n = k$, any $V(k + 1)$ operation's edge must stem from a vertex associated with $P(m)$, $m < k + 1$, i.e., $m \leq k$. Then as $R(k) = 0$ includes $R(m - 1) = 0$, all $V(m)$ must have done (Property 2). Meanwhile, $R(k) = 0$ includes $R(m) = 0$ as well, so all $P(m)$ must have been done to make $R(m)$ zero. According to Rule 1 and Rule 2, since any $P(m)$ operation has already performed with its V(m), the $V(k + 1)$ must have been invoked between them.

Conclusion As for $n = t$, the statement indicates that all $V(t + 1)$ have been invoked, namely all these events have happened. Notice that $R(t)$ equals to the reference count, then our protocol is verified. □

Specifically, we implement POP in the Quatrain server as follows: (1) the Quatrain Thread inserts a V operation before the actual start() runs, and the Quatrain Runnable performs a V operation in its constructor; (2) we wrap user-defined run() method with a P operation inserted after the actual run(); (3) preturn invokes a V operation before passing replies to the *responder* that does a P operation after each network output operation; (4) a final preturn is invoked after every called procedure. The corresponding final network output operation wait until the reference count equals zero.

POP's implementation is of high efficiency, since all operations are lightweight without any coarse-grained lock-wait-notify synchronization.

3.3 Exception and Failure

Quatrain is not ambitious to guarantee a transaction, but instead try best to get as more results as possible even when some remote problems happen. Actually, the multi-return RPC confines the impact of remote partial failures, since the survival results may independently become available. And the reply set offers sufficient information on the state of the corresponding call, such as an error message, whether it is still meaningful to wait, and whether the already received replies are partial or complete.

The main mechanism for failure control in Quatrain is timeout, namely the maximum amount of time a RPC would like to wait. From the user's perspective, we define a *exception* as when the server finishes the call before timeout but replies are still partial. And a *failure* means that the call is timed out. Usually, exceptions happen when the server encounters some managed internal error, while failures indicate that the server may fail or the connection is broken.

Programmers handle sever-side exceptions or errors in a similar manner as local procedures, and an error type can be sent back via `preturn`. The multi-return RPC is robust and remains graceful when exceptions or failures happen. All the unexpected states are exposed via the reply set's flags, while available partial replies, if any, are still retrievable. This property is rarely supported in other RPC frameworks, since they only have one chance to reply all.

3.4 Scalability

Scalability is a vital issue for Quatrain. It is long been discussed what is the best structure for highly concurrent servers. Event-driven implementation [27] bases the task management on event notification and queues. Meanwhile, some other practice supports thread-based style [28], leaving all management load to threads. As for the new multi-return paradigm, we implement both models, named thread-based Quatrain and event-based Quatrain (the above described architecture is event-based, and the thread-based version cancels all queues and thread pools). Our experiences are articulated in the Section 5.

4 Application and Case Study

The essential value of Quatrain is reducing the *average* delay of aggregated data, rather than the end-to-end delay (in a worst case, the last partial return of a request may determine the end-to-end delay). Quatrain is not a solution for all, but has advantages in two basic scenarios: (1) The follow-up task is data-parallel, namely individual early partial return makes sense (Sections 4.1 and 4.2). Take Corsair (Section 4.1) for example, each partial return corresponds to some messages to send, so users averagely experience lower latency to receive messages. (2) Partial data can be utilized to construct or preview the final results. Take MapReduce Online (Section 4.3) for example, preview can be made and approximate results are enough sometimes [29] (then rest of work can be cancelled). The following subsections demonstrate the wide types of applications Quatrain applies to.

4.1 Corsair: Multi-site or Peer-to-peer System

Quatrain is suitable for multi-site or peer-to-peer systems in that their data are widely dispersed, and processing a request usually requires data from various sources that are volatile. Quatrain can minimize the negative effects of stragglers.

Corsair is a production data-sharing platform deployed in 7 universities in China. With Corsair, students can join together to create a community with a dedicated storage space. Till Mar. 2011, there are already over 19k active users, and 500+ communities in Tsinghua University. A new extension we made to current Corsair systems is enhancing its social network facilities. We enable users to join groups in other universities and send phone messages to other group members.

In the new federative system, effectively collecting cross-site users' mobile phone numbers to support instant communication is a basic requirement. But any message sending may involve several other sites. A straight design is either to set up a single master or to synchronize related cross-site users' data locally. However, the current system structure is not centralized, and some site administrators argue that local users' personal information can be queried but should not be held on any other site for security and privacy concerns. Another solution is to forward each message to its receivers sites and avoid data aggregation. But not all sites have the messaging component as it involves telecom business.

Limited by these non-technology factors, data and computation placement hardly get optimized in our practice. So our final choice is to collect data via multi-return RPC, which largely cuts off extra delay caused by waiting for some long-distance data transmission or slow nodes, and overlap data query with message sending. The average response time is remarkably reduced as evaluated in Section 5.2.

4.2 Glickr: Mashup Services and SOA

Mashup [30], [31] is a new form of web development, which construct new applications based on existing services. It usually involves aggregation of data from various sources. This architecture illustrates the design pattern of service-oriented architecture (SOA) [32], and RPC is one of the choices to realize service-oriented computing. The multi-layered services and complex inter-service dependencies all magnify the negative impact of the network latency and outlier response time.

Meanwhile, the "server push" technology [33] enables multi-return RPC in web application. Although current Quatrain have not yet implemented a JavaScript client, our serialization component [25] fully supports it, and already has a standard-RPC version of JavaScript client.

Glickr is a mashup service combining Google Earth API[2] and Flickr API[3]. Google Earth APIs display a 3D digital globe in web browsers, and Flickr APIs provide location information of photos. Taking advantage of these unique functions, Glickr displays the photo album in a novel way - placing photos on their actual locations in the globe. When you search, for example, "tower", all related photos just sit on where the towers locate.

We suppose that our photo services include many local photo search engines to provide better results on specific countries. When our mashup collects photo information from different services and reply them to the client, Quatrain plays its important role in delivering available results as soon as possible and preventing stragglers from affecting others. Prototype of the core Glickr functionality and its evaluation is introduced in Section 5.3.

4.3 Pipelined MapReduce

MapReduce Online [29] extends the popular computing framework MapReduce [9] by breaking the "barrier" that any reduce task does not begin until all map tasks have finished. This work enables snapshot preview, continuous queries, and other attractive features.

2. http://code.google.com/apis/earth/
3. http://www.flickr.com/services/api/

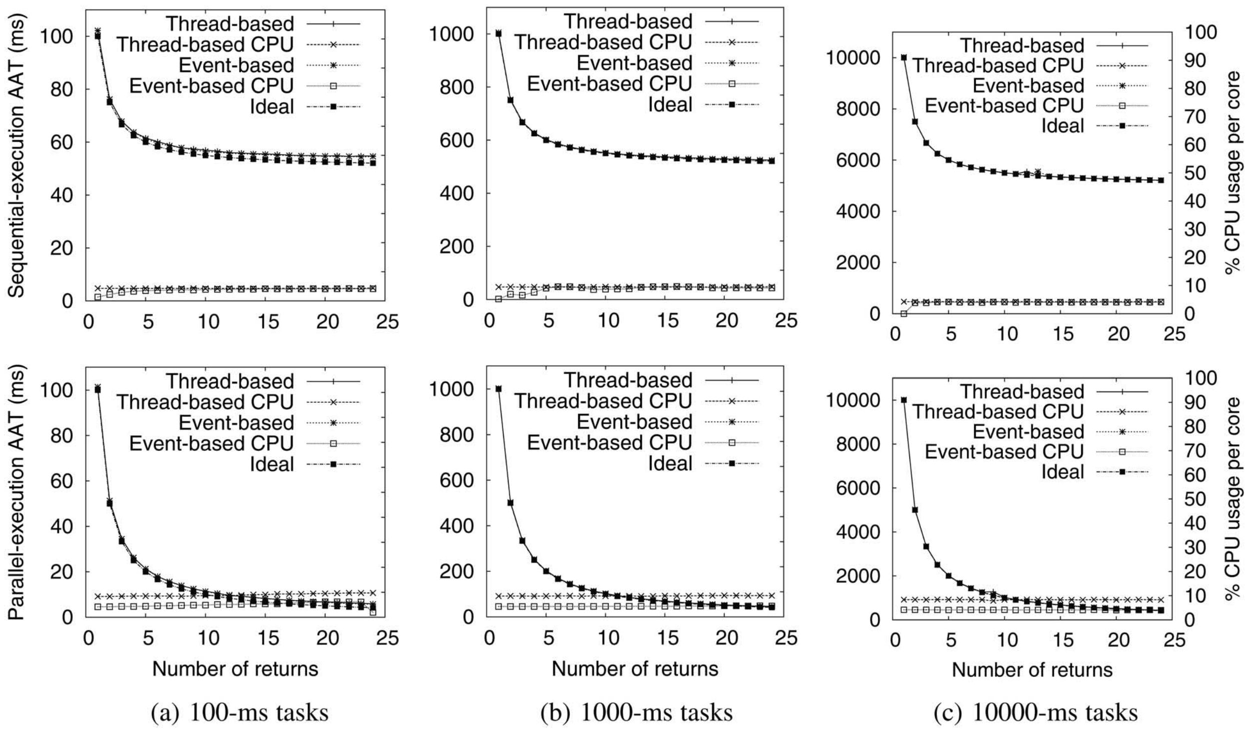


Fig. 6. AAT under low stress over the number of parts that a task is divided into.

In implementation, MapReduce Online meets several obstacles. (1) In the original design, reduce tasks invoke RPC to *pull* data from map tasks, which is not suitable for pipelining. (2) If map tasks are modified to *push* data to reduce tasks, there may be not enough slots for all corresponding reduce tasks. (3) Network I/O operations my block the map task since the map function is invoked in the same thread.

Their solution makes `pull` and `push` operations interweaved. When some reduce tasks are not scheduled due to limited slots, map tasks store the intermediate data into disks, and afterwards these behind reduce tasks will pull data in traditional manner. And they specifically solve the I/O blocking problem by running separate threads.

Actually, such logic is more naturally expressed with Quatrain. Whenever a reduce task is available, it invokes multi-return RPCs to corresponding map tasks, just like the `pull`. On the other hand, whenever intermediate data are deliverable, either from local storage or from newly produced buffer, the map task uses `preturn` to make a partial reply, just like the `push`. And there is naturally no problem on I/O blocking, since the `preturn` is not blocked. Original tricky details are all easily handled by Quatrain in a more elegant manner, which illustrates the true value of Quatrain. We have realized a Quatrain-based MapReduce with evaluation in Section 5.4.

5 Evaluation

Evaluation of Quatrain starts with micro benchmarks that analyze performance of individual RPC under both light and heavy workloads. Then we evaluate three practical applications based on Quatrain: multi-site Corsair (Section 5.2), Glickr (Section 5.3) and pipelined MapReduce (Section 5.4). The three experiments separately emphasize on the realistic network environment, the realistic service response time, and the realistic computing cloud environment.

Since the traditional definition of response time assumes only one response for a request and does not cover the situation when the returned data records arrive at different times, we define the average arrival time (AAT) as our metrics. This measurement calculates the average value of delays between the request time and each returned data record's arrival time.

5.1 Micro Benchmarks

Our micro benchmarks evaluate the overheads of Quatrain and compare thread-based as well as event-based implementations. We deploy two dedicated physical servers to separately simulate the client(s) and the server in concept. The two physical servers have identical configurations: 4 Intel(R) Xeon(R) CPUs X5660 (6 cors, 2.80 GHz), 24G memory (1.3 GHz), 1 Intel(R) PRO/1000 network connection (1000 Mbps); Ubuntu Sever 11.04 64-bit, Oracle/Sun JDK 7.

The server provides two procedures for remote calling: one executes a task in a single thread (sequential execution, SE), while the other executes it in parallel (parallel execution, PE). Tasks have a fixed total execution time $t = 100$ ms, 1 s, or 10 s. A task is evenly divided into n parts. Accordingly, each part prepares a proper number of data records as a partial reply, sleep for a proper mount of time (t/n), and invokes `preturn`. When a task only uses a single `preturn`, the results approximate performance without multi-return.

To simulate low stress, we invoke RPCs one by one to the server and calculate AAT. Fig. 6 reports the results over the number of parts that a task is divided for `preturns`. We can see that both event-based and thread-based implementations approach the ideal situations (lines almost overlap), with acceptable CPU overheads brought by multi-return

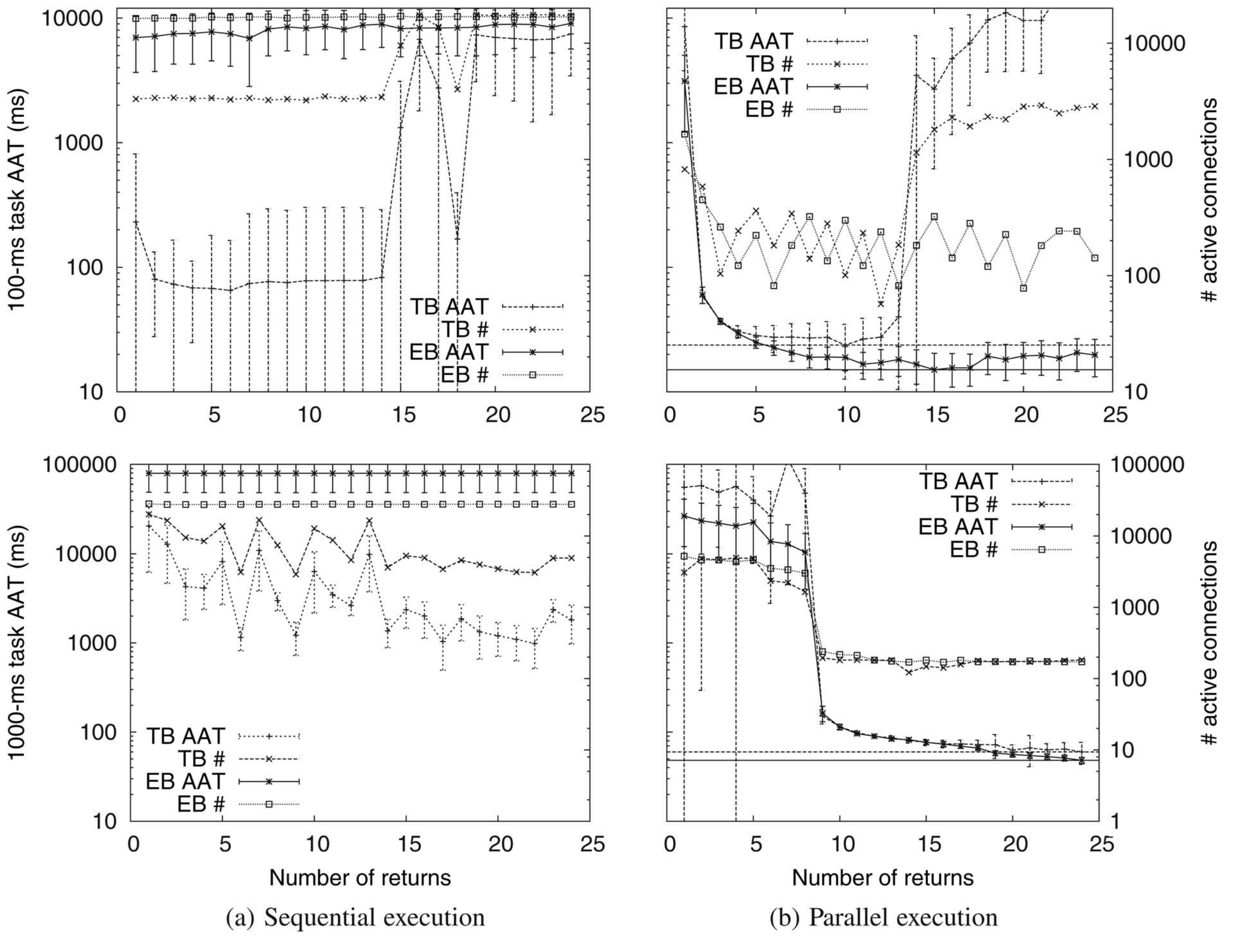


Fig. 7. AAT under high stress over the number of parts that a task is divided into. "TB/EB" stands for thread-based/event-based Quatrain. In (b), level lines show the best performance of either TB or EB. Error bars show the standard deviation of each sample.

mechanisms (mostly less than 5% for event-based Quatrain). Generally, the event-based Quatrain brings lower overheads than the thread-based version.

To simulate high stress, we start numerous parallel calling threads to reach a specified number of requests per second. Fig. 7 (a) reports results on sequential execution mode, and we can see that the thread-based Quatrain largely outperforms the event-based one, for the reason that sequential tasks take threads for long and many tasks are overstocked in queues.

Fig. 7 (b) shows performance of parallel execution. As the tasks are more fine-grained, event-based implementation shows advantages of queues. Highly concurrent tasks are efficiently handled with controlled thread management overheads. Especially, when 100-ms tasks are divided into over 12 parts each, the response time of thread-based Quatrain sharply raises due to thread overheads.

Based on these experimental results, we can arrive at some basic suggestions about when multi-return RPC produces most benefits and which implementation should be adopted. In a lightly loaded sever, multi-return RPC gains large improvement. In a heavily loaded server, such a gain depends on the procedure type and implementation choice: if the procedure is sequentially executed, the thread-based implementation is preferred, and if the procedure invokes parallel threads, the event-based implementation is the best. However, if the server is extremely overloaded without any margin to provide an enhanced service quality, the multi-return RPC is not proper to use as it may aggravate the overload.

5.2 Multi-site Corsair

In order to demonstrate the effectiveness of Quatrain in a realistic network environment, we deployed the multi-site Corsair (Section 4.1) on world-wide geo-distributed data centers.

We set up cloud servers on all available data center regions provided by three leading cloud service providers - Amazon EC2, GoGrid, and Rackspace - totally 10 ones separately located in Virginia US (A1), Northern California US (A2), Ireland (A3), Singapore (A4), Japan (A5), Chicago US (R1), Dallas US (R2), London UK (R3), Ashburn US (G1) and San Francisco US (G2). Ax servers are Amazon large instances with 7.5 GB memory, 4 EC2 computing units (2 virtual cores); Gx servers are GoGrid instances with 8 GB memory, 8 virtual cores(Intel(R) Xeon(R) CPU X5650, 2.67 GHz); and Rx servers are Rackspace instances with 8 GB memory, 4 virtual cores (Six-Core/Quad-Core AMD Opteron(tm) Processor 2423/2374 HE). All nods use Ubuntu 10.04/11.04(64-bit). Generally, all above settings well support our Corsair components with extra capacity. Since this evaluation investigates the latency of gathering user information for instant messaging, only related components (based on MySQL 5.1 and OpenJDK 1.6.0) were deployed on these nodes, without need for large volume storage.

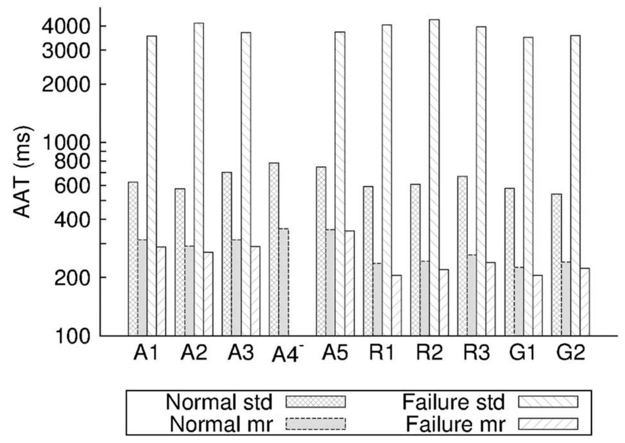


Fig. 8. AAT under normal and failure condition over nodes. "std" stands for standard RPC, and "mr" for multireturn RPC. In failure condition, A4 is shut down.

On each node, the Quatrain server exports a RPC-based API for service consumers (clients) to fetch users' mobile phone numbers of a specified group. If the group contains cross-site users, the Quatrain server has to query other peers to fetch related users' phone numbers. We record arrival times of each phone number and calculate their AAT. Actually, this is a typical example where parallel RPC is not applicable, as the cross-site users and their nodes are instantaneously calculated on the server side and are not known in advance by the clients (who consequently cannot decide the nodes to call in parallel).

The single-node configuration is set according to the actual running instance in our university, with 19,366 users, 512 communities, and 24,253 user-community relations. We suppose that a community has cross-site users from an average of 5 other nodes. All databases are filled with random data.

For each node, 100 groups are selected and sequential requests are made for each group. Fig. 8 compares AATs in standard parallel mode and multi-return mode. The left-side two columns of each cluster display the results in "normal" condition without any node failure. Considering queries on each node as an independent test case, the geometric mean value of the relative improvements is 55.7%, with maximum 61.0% and minimum 49.5%.

Furthermore, we evaluate the AAT improvements when there is one failure among nodes. We shut down a MySQL process so that requests sent to this node are all timed out (over 8 s). In Fig. 8, the right-side two columns of each cluster display the results in node-failure condition. The AAT reduction remarkably increases as the standard RPCs are largely delayed until the timeout.

Further statistics are summarized in Table 1. In normal condition, the average response time of multi-return queries is only 43.9% that of standard queries, effectively improving users' experiences on message instantaneity. In case of failure or straggler, standard queries may reach several seconds, which is hardly acceptable by the instant messaging requirement. Meanwhile, multi-return queries remain confined below 400 ms, enabling instant messaging even on hostile conditions.

TABLE 1
Corsair Performance under Different Conditions

Configuration		AAT (ms)		
		min	max	(geo)mean
Normal	standard	540.4	783.8	642.0
	multi-return	226.2	357.4	283.8
	% mr / std	39.0	50.5	43.9
Failure	standard	3494.6	4306.8	3831.6
	multi-return	205.5	349.3	254.7
	% mr / std	5.1	9.4	6.6

Note: "geo-mean" stands for the geometric mean value, and only applies to rows with relative values. "Failure" refers to the condition when A4 is shut down.

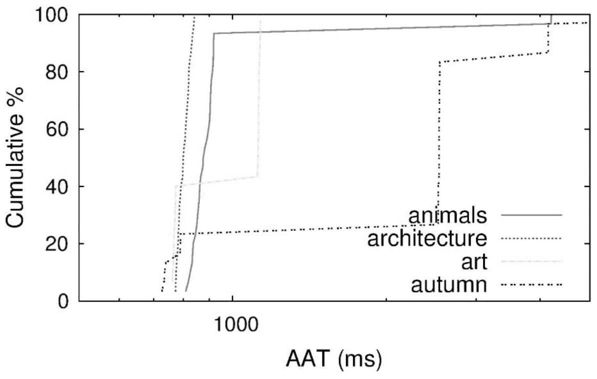


Fig. 9. Cumulative distribution of response times for individual keywords.

5.3 Mashup Glickr

To reflect performance of Quatrain over realistic service response time, we evaluate Glickr (Section 4.2) based on the widely acknowledged service Flickr.

Glickr is designed as a service aggregator that combines local photo search engines to display required photos on a virtual globe, like Panoramio[4]. Here we only measure the core process of aggregation, and compile statistics of AAT in retrieving required photos' information. In Flickr, there are many geography groups where people share photos of specific places, such as the USA, France, and China. We utilize the group search to simulate multiple local search engines.

We collect 100 most common nouns[5] (e.g., cat, lake) as keywords, and send sequential requests to the aggregator server, which then query all local search engines in parallel to retrieve related photos and their location information. We recorded arrival time of each expected photo location and compare the AAT of standard parallel calls and multi-return ones. The hardware setting is identical to the micro-benchmarks experiment in Section 5.1.

Fig. 9 depicts the response times of all local search engines for some keywords. We can see that the cumulative distribution of AATs for a specific keyword goes through several "stairs", each of which denotes a relative cluster of response times. Intervals between "stairs" show obvious time early results waste on waiting for later ones. Moreover, the long

4. http://www.panoramio.com
5. http://www.flickr.com/photos/tags/

TABLE 2
Comparison of Standard-Parallel-Mode and Multi-Return-Mode Glickr

AAT (ms)	min	max	(geo)mean
Standard	842.0	9843.7	2808.3
Multi-return	831.6	3287.7	1433.8
% mr / std	12.0	287.3	58.7

Note: "geo-mean" stands for the geometric mean value, and only applies to rows with relative values.

tails of some curves (eg. "animals", "autumn") indicate the existence of stragglers who may drag the whole response time. Note that our word choice is not biased, and they are typical according to our observation on other words. The whole set of response time values cover 3 orders of magnitude. In addition, basic statistics are listed in Table 2. On average, the multi-return mode costs only 58.7% time of the traditional parallel mode, thanks to Quatrain's tolerance for stragglers and response time diversity.

Based on such realistic characteristics, the two versions of aggregation service separately based on standard-parallel and multi-return RPC perform differently as shown in Fig. 10. Since AAT values are somewhat in random manner, we order keywords on x tics by their multi-return AATs, in order to better show what proportion of standard-parallel-mode values are above (slower than) the multi-return values. The average improvement is over 41%, thanks to the multi-return tolerance for stragglers and diversity. Recall the experiments in Google and Amazon [15] mentioned in Section 1. This level of improvement is highly valuable for real services.

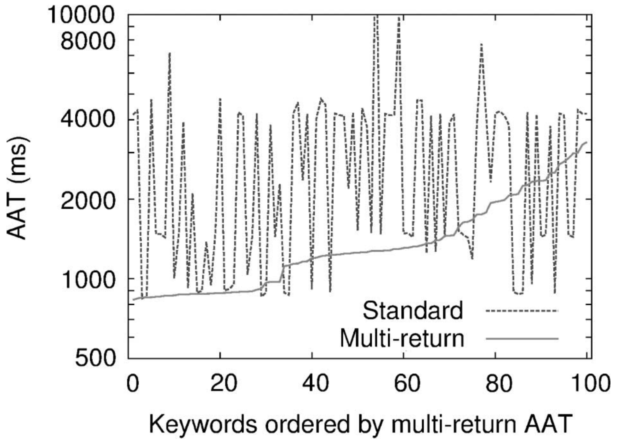


Fig. 10. AAT of aggregation services over ordered keywords.

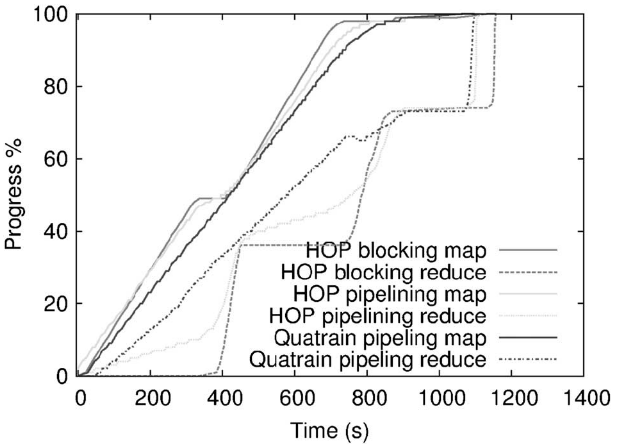


Fig. 11. Progress of map and reduce tasks over time.

5.4 Pipelined MapReduce

In order to evaluate Quatrain's usability and scalability in large-scale computing environment, we modify the implementation of MapReduce Online (Section 4.3) named HOP[6], and replace its original data transfer mechanism with Quatrain.

We launched 100 high-CPU medium instances on Amazon EC2 to carry out this evaluation. Each node has 1.7 GB memory, 5 EC2 compute units (2 virtual cores), and moderate I/O Performance. According to the paper of MapReduce Online, we choose the same application WordCount in its package. Data are stored on HDFS with 512 MB blocks, and 2 replica.

We configure 400 map tasks and 50 reduce tasks to count 200 GB data, and use the HOP progress report to observe map and reduce tasks. Fig. 11 shows the progress results of three MapReduce types: the blocking mode of HOP, the pipelining mode of HOP and the pipelining mode of Quatrain-based HOP. As we can see, Quatrain multi-returns the map intermediate data and speeds up reduce tasks.

Although end time is not largely improved due to the sorting phase, the raised reduce progress enables features of MapReduce Online, such as snapshot preview. According to [29], a Top-K query that finds 20 most frequent words get the final results by preview using only 60% of the time to finish the whole task. This illustrates the meaning of the raised progress curve by Quatrain. Moreover, the total data transfer via Quatrain is over 60 GB through 100 nodes, illustrating the scalability of Quatrain.

6. http://code.google.com/p/hop/

6 RELATED WORK

A similar multi-return pattern [20] has appeared in some modern programming languages, e.g., "yield return" in C# [21] and "yield" in Python [22]. However, their usage is highly confined and far less flexible than `preturn` in Quatrain. For example, the C# yield statement can *only* be used inside an iterator block. More essentially, the yield mechanism works by repeatedly pending and resuming the called procedure, fundamentally different with the parallel remote execution in Quatrain. By contrast, `preturn` achieves similar concise syntax as yield, but well supports multi-threaded and distributed programming.

An optimization for data aggregation similar to Quatrain has been done in FlumeJava [23], where an invocation `emit()` can be decoupled and fused into follow-up operations to avoid waiting slow peers. However, this mechanism resides in a specific programming model and therefore lacks generality. Furthermore, FlumeJava and other data/execution flow optimizers (e.g., DryadLINQ [24]) are not compatible with traditional layers abstraction (that means existing services and interfaces need architectural reconstruction to adopt such data/execution flow optimization) and therefor not suitable for pervasive service-based systems (they can only apply internally to a single service within such a system). Although Quatrain does not provide comprehensive optimization as they do, it is easy-to-use and more generally applicable. Systems do not need architectural reconstruction to adopt Quatrain.

Asynchronous RPC uses non-blocking invocation. Some implementations [34], [35] even do not support reply; some others enable clients to retrieve replies later by querying specific objects, such as Promises [36]. Quatrain also returns immediately and incrementally offers replies, but the essential difference is that asynchronous RPC still treats the replies as a whole. In addition, asynchronous RPC aims at parallelism, rather than lowest response time. Since Quatrain's multi-return RPC can work in almost a blocking manner when obtaining the next data element, we can regard multi-return RPC as semi-asynchronous.

More recently, RPC Chains [37] propose a new communication primitive for a serious of sequential RPCs to several servers, so that the intermediate data can directly flow between servers instead of involving the client in every call. Their work applies to chained RPCs while ours to nested ones.

Finally, we should clarify the relationship between Quatrain and the message passing system, such as Amazon Simple Queue Service (Amazon SQS) [38]. From high level of view, remote calling frameworks are mostly based on some form of message passing system. Actually, Quatrain can use Amazon SQS as its underlying communication infrastructure instead of sockets. For end programmers, it requires more labor to directly use low level communication techniques. In that case, programmers have to directly handle most intricate issues in Quatrain implementation as discussed in Section 3.

7 Conclusion

Quatrain provides a new angle to address the delay-by-layers issue. As far as we know, it is the first practical framework that realizes a multi-return programming paradigm for distributed systems. We contributed the initial simple semantics, underlying partially ordering protocol, and implementation experiences of both thread-based and event-based structures.

Our work illustrates, through theory as well as practical applications, the four main benefits of multi-return paradigm: early data arrival, workflow pipelining, straggler/fault tolerance, and support for potential next-generation data center networks.

We expect wide implementation and application of the multi-return paradigm. Our implementation in Java and all experiment data are open-source via http://github.com/stanzax (codename: Quatrain). Moreover, our key ordering protocol is designed friendly to language level extensions which may bring higher efficiency and more convenience for users. We leave this for future work.

Acknowledgment

We greatly appreciate the valuable support from the following people: Bingyao Ma offered his commercial version of Hprose RPC whose serialization/deserialization components are directly adopted by Quatrain; Yang Zhang wrote the initial version of Glickr; Likun Liu provided wise advices to our work. This work was supported by National Basic Research (973) Program of China (2011CB302505), Natural Science Foundation of China (61170210), and National High-Tech R&D (863) Program of China (2012AA012600).

References

[1] A. O. Ramirez, "Three-tier architecture," *Linux J.*, vol. 2000, Jul. 2000.
[2] G. T. Heineman and W. T. Councill, *Component Based Software Engineering: Putting the Pieces Together*. Boston, MA: Addison-Wesley Professional, Jun. 2001.
[3] M. Stal, "Web services: Beyond component-based computing," *Commun. ACM*, vol. 45, pp. 71–76, Oct. 2002.
[4] E. Pitt and K. McNiff, *Java.rmi: The Remote Method Invocation Guide*. Boston, MA: Addison-Wesley, 2001.
[5] S. McLean, K. Williams, and J. Naftel, *Microsoft. Net Remoting*. Redmond, WA: Microsoft Press, 2002.
[6] The Apache Software Foundation. *Apache thrift* [Online]. Available: http://thrift.apache.org/, accessed on Dec. 2012.
[7] S. Seely, *SOAP: Cross Platform Web Service Development Using XML*. Upper Saddle River, NJ: Prentice Hall, 2001.
[8] L. Richardson and S. Ruby, *RESTful Web Services*, 1st ed. Sebastopol, CA: O'Reilly Media, May 2007.
[9] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Operat. Syst. Des. Implement.*, 2004, vol. 6, p. 10.
[10] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center TCP (DCTCP)," in *Proc. ACM Conf. SIGCOMM (SIGCOMM '10)*, 2010, pp. 63–74.
[11] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken, "The nature of data center traffic: Measurements & analysis," in *Proc 9th ACM SIGCOMM Conf. Internet Meas. (IMC '09)*, 2009, pp. 202–208.
[12] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proc. 23rd ACM Symp. Operat. Syst. Princ. (SOSP '11)*, 2011, pp. 29–41.
[13] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. 8th USENIX Conf. Operat. Syst. Des. Implement. (OSDI'08)*, 2008, pp. 29–42.
[14] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. 9th USENIX Conf. Operat. Syst. Des. Implement. (OSDI'10)*, 2010, pp. 1–16.
[15] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: Listen to your customers not to the hippo," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining (KDD'07)*, 2007, pp. 959–967.
[16] Internap. *Latency: The achilles heel of cloud computing* [Online]. Available: http://www2.internap.com/wp/latency-the-achilles-heel-of-cloud-computing, accessed on Nov. 2010.
[17] E. C. Cooper, "Replicated distributed programs," in *Proc. 10th ACM Symp. Operat. Syst. Princ. (SOSP'85)*, 1985, pp. 63–78.
[18] M. Satyanarayanan and E. H. Siegel, "Parallel communication in a large distributed environment," *IEEE Trans. Comput.*, vol. 39, no. 3, pp. 328–348, Mar. 1990.
[19] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *Proc. ACM SIGCOMM Conf. (SIGCOMM'11)*, 2011, pp. 50–61.
[20] O. Shivers and D. Fisher, "Multi-return function call," in *Proc. 9th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP '04)*, 2004, pp. 79–89.
[21] MSDN. yield (c# reference) [Online]. Available: http://msdn.microsoft.com/en-us/library/9k7k7cf0.aspx, accessed on Dec. 2012.
[22] Python reference manual. The yield statement [Online]. Available: http://docs.python.org/release/2.5.2/ref/yield.html, accessed on Dec. 2012.
[23] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum, "Flumejava: Easy, efficient data-parallel pipelines," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI'10)*, 2010, pp. 363–375.

[24] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proc. 8th USENIX Conf. Oper. Syst. Des. Implement. (OSDI'08)*, 2008, pp. 1–14.

[25] Andot. Open hprose. [Online]. Available: http://github.com/andot/oh/

[26] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.*, vol. 2, pp. 39–59, Feb. 1984.

[27] M. Welsh, D. Culler, and E. Brewer, "SEDA: An architecture for well-conditioned, scalable internet services," in *Proc. 18th ACM Symp. Oper. Syst. Princ. (SOSP'01)*, 2001, pp. 230–243.

[28] R. von Behren, J. Condit, and E. Brewer, "Why events are a bad idea (for high-concurrency servers)," in *Proc. 9th Conference Hot Topics Oper. Syst.*, 2003, vol. 9, p. 4.

[29] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc. 7th USENIX Conf. Netw. Syst. Des. Implement. (NSDI'10)*, 2010, p. 21.

[30] X. Liu, Y. Hui, W. Sun, and H. Liang, "Towards service composition based on mashup," in *Proc. IEEE Congr. Serv*, 2007, pp. 332–339.

[31] D. E. Simmen, M. Altinel, V. Markl, S. Padmanabhan, and A. Singh, "Damia: Data mashups for intranet applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data (SIGMOD'08)*, 2008, pp. 1171–1182.

[32] T. Nestler, "Towards a mashup-driven end-user programming of SOA-based applications," in *Proc. 10th Int. Conf. Inform. Integr. Web-based Appl. Serv. (WAS'08)*, 2008, pp. 551–554.

[33] Wikipedia. *Push Technology* [Online]. Available: http://en.wikipedia.org/wiki/Push_technology/, accessed on Dec. 2012.

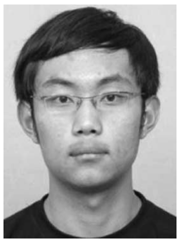
[34] G. A. Champine, D. E. Geer, Jr., and W. N. Ruh, "Project Athena as a distributed computer system," *Computer*, vol. 23, pp. 40–51, Sep. 1990.

[35] S. Microsystems, "RPC: Remote procedure call protocol specification version 2," *Internet Netw. Working Group Request Comments (RFC)*, no. 1057, 1988.

[36] B. Liskov and L. Shrira, "Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implement. (PLDI'88)*, 1988, pp. 260–267.

[37] Y. J. Song, M. K. Aguilera, R. Kotla, and D. Malkhi, "RPC chains: Efficient client-server communication in geodistributed systems," in *Proc. 6th USENIX Symp. Netw. Sys. Des. Implement. (NSDI'09)*, 2009, pp. 277–290.

[38] Amazon Web Services. Amazon Simple Queue Service. [Online]. Available: http://aws.amazon.com/sqs, accessed on Dec. 2012.



Jinglei Ren received the BE degree from Northeast Normal University, Changchun, China, in 2010. He is currently a PhD candidate in Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include distributed systems, storage techniques, and operating systems. He has developed a storage system for virtual machines with enhanced manageability, and is currently working on the flash-aware and energy-saving smartphone filesystem.

Yongwei Wu received the PhD degree in applied mathematics from the Chinese Academy of Sciences, Beijing, China, in 2002. He is currently a professor in Computer Science and Technology at Tsinghua University, Beijing, China. His research interests include distributed processing, virtualization, and cloud computing. Dr. Wu has published over 80 research publications and has received two Best Paper Awards. He is currently on the editorial board of the *International Journal of Networked and Distributed Computing* and *Communication of China Computer Federation*. He is a member of the IEEE.

Meiqi Zhu received the BE degree from the College of Information Technical Science, Nankai University, China, in 2010. He is pursuing Master's degree in the Department of Computer Science and Technology, Tsinghua University, Beijing, China. His research interests include distributed computing and storage systems.

Weimin Zheng received the BS and MS degrees from Tsinghua University, Beijing, China, in 1970 and 1982, respectively, where he is currently a professor of Computer Science and Technology. He is the research director of the Institute of High Performance Computing at Tsinghua University, and the managing director of the Chinese Computer Society. His research interests include computer architecture, operating system, storage networks, and distributed computing. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.