Efficient Session Type Guided Distributed Interaction

KC Sivaramakrishnan, Karthik Nagaraj, Lukasz Ziarek and Patrick Eugster

Purdue University, West Lafayette IN, USA
{chandras,knagara,lziarek,peugster}@cs.purdue.edu

Abstract. Recently, there has been much interest in multi-party session types (MPSTs) as a means of rigorously specifying protocols for interaction among multiple distributed participants. By capturing distributed interaction as series of typed interactions, MPSTs allow for the static verification of compliance of corresponding distributed object programs. We observe that explicit control flow information manifested by MPST opens intriguing avenues also for performance enhancements. In this paper, we present a session type assisted performance enhancement framework for distributed object interaction in Java. Experimental evaluation within our distributed runtime infrastructure illustrates the costs and benefits of our composable enhancement strategies.

1 Introduction

Interaction between software components is one of the fundamental concerns of software development, and yet precisely describing the interactions between components remains a difficult endeavor. Real-world distributed systems involve multiple remote components, independently communicating messages and coordinating activities among one another. Web services, e-commerce applications, and protocols like SMTP, POP3, are just some examples for structured protocols involving multiple interacting peers. Such interactions are usually implemented using message transfer over reliable socket communication. Unfortunately, such low level communication neither offers type safety on messages nor the ability to specify and statically type check communication protocols, making development of distributed software difficult.

Session types have been proposed as a way to precisely capture *complex* interactions between peers [14]. They describe interaction protocols by specifying the type of messages exchanged between the participants. Implicit control flow information such as branching and loops can also be enumerated. Session types were originally envisioned for languages closely based on process calculi, and initially used for specifying bi-party interaction. They have since been extended to *multi-party session types* (MPSTs) [10]. Explicit control flow information manifest in MPSTs, opens intriguing avenues for global performance enhancement of distributed multi-party interaction. In this paper, we present a session type assisted protocol enhancement framework for optimizing distributed object interaction in Java. To illustrate our approach, consider the example in Fig. 1,

which describes a simple protocol implemented in Java RMI, to invite co-workers obtained from a social networking database to a party.

```
1  void inviteCoworkers() {
2    Event evt = me.createEvent("Party", date);
3    Employer myEmp = me.getEmployer();
4    Location myLoc = me.getLocation();
5    for (Member mbr : me.getFriends()) {
6      if (myEmp.equals(mbr.getEmployer()) &&
7          myLoc.equals(mbr.getLocation()))
8          mailSvr.sendMail(mbr.getEmailAddress(), evt);
9    }
10 }
```

Fig. 1: Client implementation to invite co-workers to a party

Abstractly, the client iterates through a list of friends sending an email invitation to those who are employed by the same employer and work at the same location. In this example, `me`, `mbr` and `mailSvr` are remote objects, therefore, all of the method invocations on these objects are done remotely requiring a network round trip. In a high latency network the client, unfortunately, would spend most of its time stalled on responses from the server. In this manner, excessive remote method invocations quickly lead to serious scalability and performance concerns.

A standard technique to overcome the network delays is to structure code based on *data transfer objects* (DTOs) or *remote facade* patterns [6]. Such patterns advocate that new remote interfaces be defined in the `Member` class specifically to invite co-workers to an event. Although such a specialized definition can reduce the number of remote method invocations (RMI), it is neither composable nor extensible, as new features would require further specialization. To make matters worse, specialization is not even possible if an RMI call depends on local operations. Previous work has looked at alleviating some of these costs through various techniques, but are limited to bi-party interaction, semantically restricted to part of the protocol [12], or require code to be written in a style amenable to enhancement [13].

Guided by the MPST information, we introduce *combined* type and compiler driven performance enhancements. Our approach extends the previously described bi-party enhancements to multiple participants, and, more importantly, seamlessly composes enhancements techniques.

Fig. 2 illustrates the abstract description of the invitation protocol for the example described in Fig. 1. The programmer defines a new global session type [10] through the use of a **`protocol`** block. This block explicitly defines the participants of the protocol, the types of messages exchanged between the participants, and the order in which the messages are exchanged. Messages are sent asynchronously between the participants. We use, `client:` **`begin`** to express that the client initiates the protocol. Each message in the protocol has a syntax `A->B: <T>`, defining that the participant `A` sends to participant `B` a message whose type is `T`. For each friend on the friends list, the infoSvr sends the member information to the client. This is represented by a recursive type,

```
1  protocol invitation {
2    participants client, infoSvr, mailSvr;
3    client: begin;
4    infoSvr->client: <Employer>;
5    infoSvr->client: <Location>;
6    infoSvr:
7      [infoSvr->client: <Member>;
8       infoSvr->client: <Employer>;
9       infoSvr->client: <Location>;
10      infoSvr->client: <EmailID>;
11      client:
12       {INVITE: client->mailSvr: <EmailID, Event>,
13        NOOP: }
14     ]*
15 }
```

Fig. 2: Global session type of invitation example

`infoSvr:[...]*` (see lines 6 to 14). `infoSvr` is the loop guard in the recursive type since it decides whether the next iteration of the loop is executed. Based on the location and employer, the client chooses to send an email invitation (see lines 11 to 13). Notice that the protocol is abstract in both the event and who is invited. The actual implementation of the protocol is specified by the client. The participants can be statically verified for conformance.

It is evident from the session type of the invitation protocol that the first two messages from the `infoSvr`: `Employer` and `Location`, can be batched together as a single message. Similarly, the first four messages in the recursive type are batched together. However, is it possible to batch multiple iterations of the recursive type together? This is less clear. First, we must assert that the `INVITE` message sent by the client to the `mailSvr` does *not* influence the next iteration of the `infoSvr`. Since MPSTs explicitly defines all remote interactions, we can statically assert that there is no causal dependence between message sent to `mailSvr` and subsequent iteration decisions of `infoSvr`. With this knowledge, the code can be rewritten such that all of the friend's information is sent to the client in one batch. The client sends the `mailSvr` a batch of email addresses of friends who are to be invited to the party. Thus the entire protocol is performed in two batched network calls, while still adhering to the protocol specification.

In this paper, we also study the interaction, composability, and performance of enhancement strategies for interacting distributed objects. To our knowledge this is the first work that utilizes session types for global enhancement of interaction. In summary, the main contributions of the paper include:

- A Java extension that integrates MPSTs (Sec. 2).
- A detailed study of performance enhancements in the presence of a global interaction protocol (Sec. 3).
- An empirical analysis of optimization strategies in a prototype framework (Sec. 4).

2 Programming with Session Types

Global session types [10] in a multi-party session provide a universal view of the protocol involving all of the participants. For every properly defined session type, there exists a local view of the global session type, called the *local session type*. A projection from a global session type to such a local session type is well-defined. Local session types for the invitation protocol projected from the global session types in Fig. 2 are shown below. The local types are, indeed, very similar to global types except that only those actions which influence the participant appear in the participant's local session type. Message sends and receives are explicit in the local session type as is the order in which they are performed.

In our system, the programmer defines the global session types, and the coresponding local session types are automatically generated. The programmer implements each participant code such that they conform to their corresponding local types.

```
protocol invitation@infoSvr {
 client: ?begin;
 client: !<Employer>;
 client: !<Location>;
 ![client: !<Member>;
   client: !<Employer>;
   client: !<Location>;
   client: !<EmailAddr>]* }

protocol invitation@mailSvr {
 client: ?begin;
 infoSvr:
 ?[client:
  ?{INVITE: client: ?<EmailAddr,
                     Event>,
    NOOP: }]* }
```

```
protocol invitation@client {
 !begin;
 infoSvr: ?<Employer>;
 infoSvr: ?<Location>;
 infoSvr:
 ?[infoSvr: ?<Member>;
   infoSvr: ?<Employer>;
   infoSvr: ?<Location>;
   !{INVITE: mailSvr:!<EmailAddr,
                        Event>,
     NOOP: }]* }
```

Fig. 3: Local session types for client, infoSvr and mailSvr

The local type for the `infoSvr` is given in **`protocol`** `invitation@infoSvr`. `client:?begin` indicates that this protocol is initiated by the client. Message sends are represented by `A: !<T>`, defining that the local participant sends to participant `A` a message of type `T`. Conversely, `B: ?<T>` shows that the local participant waits to receive a message of type `T` from participant `B`. The syntax `![...]*` represents that this participant controls the loop iteration while all the participants with `A: ?[...]*` execute the next iteration of the loop only if `A` chooses to execute the next iteration. Similarly, syntax `!{L1:T1,...}` states that this participant chooses one of the set of labels `{L1,...}` to execute and other participants with `A: ?{L1:T1',...}` also execute the same label.

The **`protocol`** keyword is used to define the global session type, which is registered with the *session registry* (similar to an RMI registry). After the protocol has been registered, a host can register as a participant by creating a *session socket*. A session socket allows a host to act as a participant and communi-

cate with other participants as dictated by the protocol. Communications on the session socket are type checked using the corresponding local type to ensure that the participant adheres to the protocol. The programmer can create a new protocol instance using the `instantiate` command, with a string representing the instance name. This name must be unique for each instance of the protocol in a session registry. Thus multiple instances of the same protocol can execute concurrently.

```
SessionRegistry.instantiate(invitation, "i");
```

The session registry prepares session sockets for each participant. A host wishing to participate in the invitation protocol as a client would request the session registry as shown below. Here we create a new session socket `ss` reflecting the type `invitation@client` for the invitation protocol for the instance `i`:

```
SessionSocket ss=SessionRegistry.lookup(invitation,"i","client");
```

Once a session socket has been created, the programmer uses a *session API* to implement the actual communication and control structures. We adopt a Java syntax for session API similar to what has been described in SessionJava [11]. Fig 4 shows the mapping between protocol description syntax and session API. We extend the `send` and `receive` syntax to explicitly denote the participant who performs the matching communication. Previous work on multi-party session types explicitly creates channels for communication between the peers [10]. We assume that each participant has a unique channel to every other participant. We also assume that the participants are single-threaded. This ensures that all communication over channels is linear.

```
!begin           ss.begin()                  //initiates protocol
A: ?begin        ss.awaitBegin()             //waits for protocol begin
A: !<T>          ss.send(A, obj)             //obj is of type T
A: ?<T>          ss.receive(A)               //received obj is of type T
!{L:T, ...}      ss.outbranch(L){}           //body of case L of type T
A: ?{L:T, ...}   ss.inbranch(L){}            //body of case L of type T
![T]*            ss.outwhile(bool_expr){}    //body of outwhile of type T
A: ?[T]*         ss.inwhile(A){}             //body of inwhile of type T
```

Fig. 4: Protocol description syntax and their mapping to session API

Using the session API described above, the client described in Fig. 1 would be expressed as outlined in Fig. 5. As per the protocol description, only one participant initiates the protocol by invoking `begin`, while all other participants wait for this notification by invoking `awaitBegin`. In this example, the client initiates the protocol while the `infoSvr` and `mailSvr` wait for the request from the client. The `inwhile` loop runs an iteration if the corresponding `outwhile` loop at the `infoSvr` is chosen to run an iteration. The label choice made at the `outbranch` is communicated to the peers who wait on the corresponding `inbranch`. The peers then execute the code under the chosen label.

In order to type check the participants for protocol conformance, we assume that the protocol description is available at every participant during compilation.

```
1  void inviteCoworkers() {
2    SessionSocket ss =
3      SessionRegistry.lookup("invitation", "client");
4    ss.begin();
5    Employer myEmp = ss.receive("infoSvr");
6    Location myLoc = ss.receive("infoSvr");
7    Event evt = me.createEvent("Movie", date);
8    ss.inwhile("server") {
9      Member m = ss.receive("infoSvr");
10     if (myEmp.equals(ss.receive("infoSvr")) &&
11         myLoc.equals(ss.receive("infoSvr"))) {
12         EmailID eid = ss.receive("infoSvr");
13         ss.outbranch(INVITE) {
14           ss.send("mailSvr", eid); ss.send("mailSvr", evt);}}
15     else {ss.outbranch(PASS) {}}
16 }}
```

Fig. 5: Client implementation of invitation protocol using session API

Exceptions might be raised while instantiating protocols from the registry if the instance name is not unique or if a program tries to create multiple session sockets for the same participant. Exceptions are also raised if any of the participants of the protocol fail. A node failure results in the termination of the protocol, though we could envision a system where a participant replica could be assigned the task of continuing the protocol.

3 Performance Enhancement Strategies

We classify the enhancements into two categories based on the information required to perform them – (1) type driven enhancements and (2) compiler driven enhancement (see Fig. 6). Type driven enhancements are performed by the *type inspector*, based *only* on the information provided by the session types. Our compiler performs static analysis on the remote interaction code *in combination with* session types. Such enhancements are classified as compiler driven enhancements.

3.1 Type Driven Enhancements

Batching sends. Multiple consecutive sends to the *same* participant are batched together, provided there are no intervening receives by the sender. For example, in the session type shown in Fig. 2, the two sends at the beginning are batched together by the type inspector. These batched sends are represented in the optimized session type as:

```
server->client: <Employer, Location>
```

When the runtime system encounters the first message send of type `Employer`, instead of eagerly pushing the message out, it waits for the second message of type `Location`. A batch is created with both of these messages and sent to the server. The runtime at the server decouples the messages and passes them

individually to the program. Batching, therefore, remains transparent to the program. The type inspector also batches sends in a recursive type if the participant acting as the loop guard does not perform any receives in the body of the loop.

```
server: [server->client1: <String>;
         client1->client2: <int>]*
```

is optimized to

```
server->client1: <String>*
client1->client2: <int>*
```

where all of the messages from `server` to `client1` and from `client1` to `client2` are sent in individual batches. Notice that the optimized session type does not have a loop. The sends occurring in a loop pattern are often resolved during batch data update, when objects belonging to a collection are updated. If the loop guard performs a receive in the body of the loop, sends in the loop cannot be batched since the decision to run the next iteration of the loop might depend on the value of the message received. Consider

```
client: [server->client: <bool>]*
```

where a possible `client` implementation could be

```
ss.outwhile(ss.receive("server"));
```

Here, sends should not be batched as every receive depends on the previously received value.

Choice lifting. Choice lifting is an enhancement by which label selection in a choice is made as early as possible so that more opportunities for enhancements are exposed. Consider the following snippet of protocol description:

```
B->A: <bool>;
A: [A->B: <int>; A: {L1: A->B: <String>, L2: A->B: <bool>}]*
```

The number of messages sent in the above protocol is $1 + 2 * num\_iterations$. Participant `A` is the guard in both the recursive type and the choice. Since the boolean conditional at the choice can only depend on the last received message at `A` (which is the receive of a **`bool`** at line 1), the choice can be lifted as far as the most recent message reception. The choice can be lifted out of the recursive type since the label choice at `A` is made independent of the looping decision made at `B`. The result of the enhancement is

```
B->A:<bool>;
A: {L1: A: [A->B: <int, String>]*, L2: A: [A->B: <int, bool>]*}
```

We can further optimize the above type by batching the loop sends as described earlier. The optimized session type is given below. Notice that the optimized session type needs to perform just 2 message sends; a **`bool`** from `B` to `A` and a batch from `A` to `B`.

```
B->A:<bool>;
A: {L1: A->B: <int,String>*, L2: A->B: <int,bool>*}
```

3.2 Compiler Driven Enhancements

Data flow analysis. In client/server systems, often we encounter a pattern where a client requests items in a collection and based on some property of the item, chooses to perform an action on a subset of the items in the collection. This is analogous to the UPDATE statement of SQL where the WHERE clause describes the subset to be updated. The example described in Fig. 2 falls into this category where, based on the employer and location of the member, the client chooses to invite the member to the party. The following snippet shows the optimized local type at the server.

```
1 protocol invitation@server {
2   client: ?begin;
3   client: !<Employer, Location>;
4   ![client: !<Member, Employer, Location>;
5     client: ?{INVITE: client: ?<Event>,
6               PASS: }]*
7 }
```

The `server` is the loop guard in this example, deciding whether to execute the next iteration of the loop. At every iteration, the server might receive a value of type `Event` from the client (line 5). Session types do not tell us whether such a received value influences the boolean conditional at the loop entry or the sends in the loop. Session types gives us the control flow information, but no data flow information. Hence, we implement a flow sensitive data flow analysis in the compiler which determines whether the loop conditional or the sends are dependent on any of the receives in the loop body. This analysis is similar to the one described in remote batch invocation [12]. Session type information allows us to precisely determine the scope of the analysis. In the above example, the data flow analysis shows that neither the loop conditional nor the sends are dependent on the receives. Hence, we can optimize the session type as below, in which all of the sends are batched, followed by a second batch with all receives.

```
1 protocol invitation@server {
2   client: ?begin;
3   client: !<Employer, Location>;
4   client: !<Member, Employer, Location>*;
5   client: ?{INVITE: client: ?<Event>, PASS: }*;
6 }
```

Exporting continuations. Let us reexamine the example in Fig. 1. The client just examines the location and employer of the member profiles — all of which are located on the server — to decide on invitations. Importantly, our flow sensitive data flow analysis shows that none of the client operations depend on the local state of the client, except for the date and the type of event, which is created before any of the remote operations. In such a scenario, we execute entire client code on the server. However, this enhancement requires the corresponding fragment of the client's code to be available at the server. Such exportable pieces

of code are called *first-class continuations*. Java does not offer language support for capturing, passing, or utilizing such continuations.

Luckily, full-fledged continuations are not necessary. Instead, we leverage compiler support to statically move the continuation code to the destination. We assume that during compilation process, the whole program code is available. Thus, when our compiler determines that the invitation protocol at the client could be executed in its entirety at the server, it compiles the server with the readily available client code. Since our enhancements are performed statically, we are able to compile the remote code into the local code.

By exporting code to where the data is instead of the other way around, we can make immense savings on the data transferred over the network. This is especially beneficial for clients connected to a slow network. However, continuation exporting is impossible if the code to be exported depends on local state. Consider an extension to the invitation example of Fig. 1, where along with checking if a member works for the same employer and location of the person hosting the party, we also require user confirmation before sending out each invitation. The user intervention required at each loop iteration makes continuation exporting impossible. But batching as discussed earlier still applies since the computation is performed at the client. Continuation exporting can also hamper performance, since the computation is offloaded to a remote node. For a compute-intensive task, offloading all the computation might overload the server and thus bring down the overall performance. Our experimental results show that continuation exporting benefits thin clients and fat servers (see Sec. 4.2).

Chaining. Chaining is a technique to reduce the number of cross site RMIs. Chaining can significantly reduce end-to-end latency in a setting where the participants are Geo-distributed [13]. Consider a user shopping for music online over a 3G network. The user requests an album and buys songs with ratings higher than 8 from that album. The user then transfers the songs to his iMac. Assume that the desktop machine is connected to the Internet. The following snippet shows the pseudo code for the phone written in an RMI style.

```
1 void onlineShopping() {
2   Album a = Vendor.album ("Electric␣Ladyland");
3   for (SongInfo si : a) {
4     if (si.rating() > 8) {
5       Song s = si.buy();
6       iMac.put(s);
7 }}}
```

The corresponding session type for this protocol is given below.

```
1 protocol onlineShopping {
2   participants vendor, phone, iMac;
3   phone: begin;
4   phone->vendor: <String>;
5   vendor->phone: <Album>;
6   vendor:
```

```
7     [vendor->phone: <SongInfo>;
8      vendor->phone: <int>;
9      phone: {BUY: vendor->phone: <Song>;
10                  phone->iMac: <Song>;
11             PASS: }]*
12  }
```

Our type inspector performs batching sends enhancement on the loop with vendor as the loop guard, thereby sending all the `SongInfo` in a batch to the phone. Based on the rating, the phone downloads the song and puts it on an iMac desktop machine. Observe that songs are transferred to the iMac through the phone connected to a high latency, low bandwidth network. Chaining allows the songs to be directly transferred to iMac. Let us observe the local type of onlineShopping protocol at the phone. The following code snippet shows the local type under `BUY` label of the choice.

```
!{BUY: vendor: ?<Song>; iMac: !<Song>, PASS: }
```

The type inspector observes that the phone performs a reception of a message of type `Song` from the vendor before the sending of a `Song`. When such a potential forwarding pattern is encountered, the compiler is informed to inspect the intermediate code block to find if the intermediate computation depends on local state. If the intermediate computation is stateless, we export the continuation to the original sender, so that the data can directly be forwarded to the destination. In this case, the phone chooses to forward the song based on the rating received from the server. Hence, we export this continuation from phone to vendor, which makes the forwarding decisions and forwards the song to the desktop machine in one batch. Notice that instead of two transfers of the songs on a slow 3G network, we transfer the songs once over high-speed Internet.

One of the key advantages of our system is the ability to seamlessly combine various enhancement strategies. Notice that, the example just described effectively combines batching sends (to send songs in one batch), chaining (to skip the phone in the transfer) and continuation exporting (the logic to only choose the song with rating of at least 8).

3.3 System Design

Our system architecture is depicted in Fig. 6 – the compiler is a combination of a type inspector and Java compiler, whereas the runtime is a veneer on top of JVM. The compiler is utilized to analyze the program, discover enhancement potential, pre-generate code for enhancements, and provide an object model amenable to marshaling and serialization. Thus, the compiler generates specialized code based on our optimizations and decisions to utilize the optimizations are made at runtime.

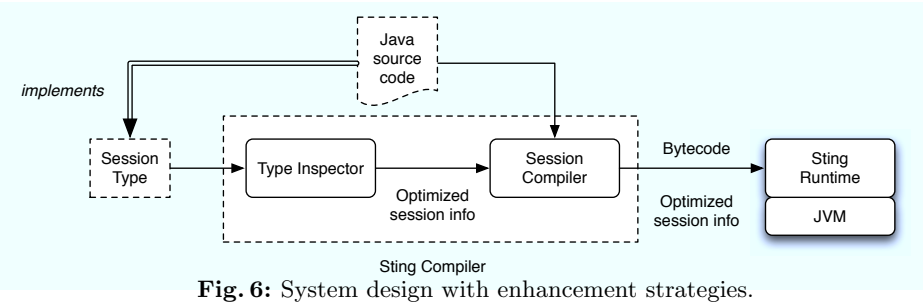

Fig. 6: System design with enhancement strategies.

4 Evaluation

We evaluate the performance gains of our optimizations from experiments that characterize typical scenarios in distributed systems. The experiments were conducted on Emulab [16] which emulates network bandwidth and latency. We used a two node setup for the batching experiments with 1Mbps link bandwidth and RTT values of 40, 80 and 150ms, inspired from ping latencies seen on the Internet. For the chaining experiments, we used 3 nodes, one of which was assigned to be the client machine with low bandwidth and high latency to the servers. The servers have Gigabit links with LAN latency interconnects. Sun Java 1.6 was used for running all our client and server Java programs. The Emulab machines were 850MHz Pentium 3 with 512MB RAM.

4.1 Batching

We study the benefits of batching sends and receives through an experiment conducted using a client-server scenario with 2 nodes. We define an operation where the client sends a cryptographic signature to the server, which verifies the signature using the public key and returns a boolean indicating success, following the session type:

```
client:
  [client->server: <Signature>;
   server->client: <boolean>]*
```

This is implemented both using a basic send and receive, and in our framework. We vary the signature size, the number of successive calls (which can be batched) and network RTT and measure the time required to complete the entire operation. As expected, the results showed that batching improves performance linearly with batch size. Batching performed better with increasing latency as the number of network round trips were reduced.

4.2 Exporting Continuation

We define two remote operations `fetchStockPrice` and `tradeStock`, which are used to fetch the current price of a *stock*, and trade a given amount of stock with the broker respectively. The client first obtains the price of the stock, performs

algorithmic trading computations on the prices and intimates the broker with the decision to buy or sell. This essentially depicts a situation where we have two remote interactions surrounding a fairly compute intensive local computation (trading). This structure of communication is commonly found in many distributed systems applications. For our experiments, we compare the basic approach with exporting continuation of trading. We export the trade computation to the remote server and batch the whole block as a single remote invocation. We ran these experiments using a server program executing on a 3GHz dual-core machine with 4GB RAM. The clients were executed on different machines with identical configuration.

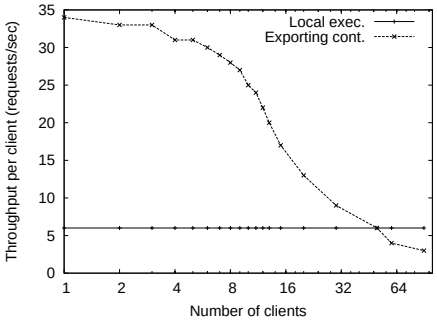

Fig. 7: Client request throughput

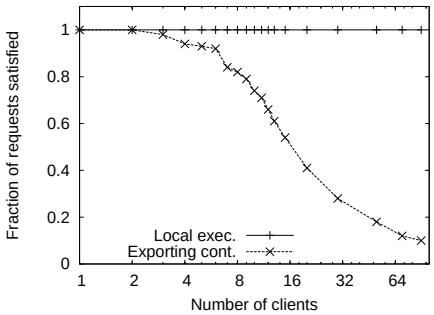

Fig. 8: Server request throughput

Client throughput. Fig. 7 shows the throughput of requests serviced per client as we increase the number of concurrent clients. The client requests are throttled at 34 requests/second. For the basic local execution of the trade on the client, the throughput achieved is independent of the number of clients. This is shown as a horizontal line with at 6 requests/sec. In this case, the critical path of computation is executed locally at the client, and hence the throughput is upper bounded by the client resources. Exporting continuation is represented by the higher throughput curve which is about 6 times larger, attributed to a powerful server. This throughput gain is understood by the ratio of computational power of the server and the client. As we increase the number of simultaneous clients, however, we see that the throughput starts dropping exponentially after about 6 clients. Note that the abscissa is in logarithmic scale.

Server throughput. Fig. 8 shows the fraction of requests satisfied. Local execution achieves a ratio of 1 shown by the straight line, because the server is under-utilized by just the remote interaction and is hence able to serve all requests received. With exported continuations, the request processing rate starts at 1 when the server is loaded on a small number of clients. As the number of clients increases, the server resources saturate and the requests are not handled at the same rate as they are received and the processing rate drops exponentially. It is important to note that the server is still able to provide higher throughput than the clients themselves, which is evident from the client throughput graph.

Server CPU utilization. Fig. 9 shows the CPU usage at the server during this experiment. About 6 parallel client requests and associated computation saturates one of the cores at 50% utilization. The remote operation is not inherently parallel and does not scale linearly on the two cores. The performance benefits beyond this point are much smaller. When the number of clients is about 50, the server CPU completely saturates and the per-client throughput equals that achieved by client's computational resources. At this point, it ceases to be worthwhile to export computation to the overloaded server for performance. This region can be estimated by comparing the performance difference between the client and the server.

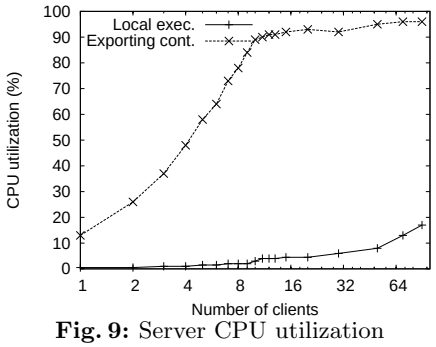

Fig. 9: Server CPU utilization

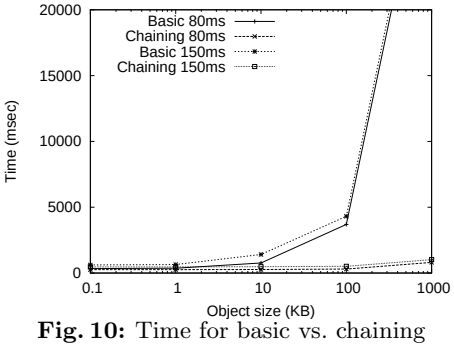

Fig. 10: Time for basic vs. chaining

Batching reduces the number of separate remote communication to one and is therefore affected only by a single RTT. Increasing the RTT would reduce the throughput of both approaches as a result of increased client latency. However, by exporting the continuation, increase in client latency is lesser and so is the decrease in throughput. As shown before, the single batched call would score ahead in large RTT settings with both network and computation benefits.

4.3 Chaining

We implemented the example of purchasing songs from the phone discussed in Sec. 3.2, where chaining is exploited to reduce communication involving the slow phone. Fig. 10 shows the comparing chained and non-chained versions. We set the network up such that the phone has a 1Mbps high latency link to the server and the PC, and the server is connected to the PC on a 100Mbps link with 2ms latency. We vary the size of the object being chained from 0.1KB to 1MB and experiment with RTT values of 80 and 150ms. The upper two curves in the graph show the basic approach used to transfer the object through the phone. In this case, the speed of transfer is limited by the link capacity of the phone's 3G network and hence we see an exponential increase in time as the size increases. However, chaining the calls has a drastic improvement in time by using the high bandwidth link to transfer the song.

5 Related Work

Session types [14] allow precise specification of typed distributed interaction. Early session types described interaction between two parties, which has then been extended to multi-party interaction by Honda et al. [10] and Bonelli et al. [3]. Honda et al. conduct a linearity analysis and prove progress of MPSTs. Linearity analysis in our system is simplified by the fact that each participant has a unique statically defined channel to every other participant, and channels are not first-class citizens. The work of Bejleri and Yoshida [2] extends that of Honda et al. [10] for synchronous communication specification among multiple interacting peers. We choose asynchronous over synchronous communication as the resulting looser coupling facilitates more aggressive optimizations.

Session types have have been applied to functional(e.g. [7])), component-based systems (e.g. [15]), object-oriented (e.g. [11]), and operating system services (e.g. [5]). Asynchronous session types for have been studied for Java [4]. Bi-party session types have been implemented in Java [11]. Our protocol description syntax is inspired from the syntax described in that work. Our framework supports multi-party protocol specification and interaction. We have also implemented a session registry to facilitate initiation of multi-party sessions. Gay et al. [8] describe how to marry session types with classes, allowing for participants to be implemented in a modular fashion. This work could be used to extend our framework.

Yeung and Kelly [17] propose runtime batching on Java RMI by performing static analysis on bytecode to determine batches. In these systems, ordering of remote and local operations influences the effectiveness of batching. Any operation performed on the result of remote method calls forces the method call to be flushed. *Remote batch invocation* (RBI) [12] performs batching on a block of code marked by the programmer. RBI reorders operations such that all remote operations are performed after the local operations and the code is exported to the server. RBI would not be able to batch a loop which requires user input on every iteration. RBI is also limited to batching specialized control structures and cannot handle arbitrary computation. Our system allows global optimization decisions to be made, and can batch across *multiple* participants.

First-class continuations [1] are a general idea to allow arbitrary computation to be captured and sent as arguments to other functions. In a distributed setting, exporting continuations are advantageous where the cost of moving data is much larger than the cost of moving computation to the data. *RPC chains* [13] reduce cross site remote procedure call overheads by exporting the callback functions to the remote host. This system requires that the user writes code in a non-intuitive *continuation passing* (CP) style [1]. Also, the callback functions cannot manipulate local state. Our system chains arbitrary code segments written in imperative style. Though we require all code to be available during compilation, our system can support separate compilation of participants, if the code were provided in CP style.

6 Conclusions

This paper is to the best of our knowledge the first to attempt to exploit session types for performance enhancements. We have shown that *combining* session types with information gathered from static analysis can yield performance improvement for distributed object interactions. We demonstrated the benefits of our approach; in particular, we have shown that our continuation exportation scheme benefits applications with thin clients and fat servers.

Acknowledgements. We are very grateful to A. Ibrahim, Y. Jiao, E. Tilevich, and W. Cook for sharing insights and the source code for RBI [12] with us.

References

1. Appel, A.W.: Compiling with Continuations. Cambridge University Press. (2007)
2. Bejleri, A., Yoshida, N.: Synchronous Multiparty Session Types. Electron. Notes Theor. Comput. Sci. 241, 3–33 (2009)
3. Bonelli, E., Compagnoni, A.: Multisession Session Types for a Distributed Calculus. In: In TGC07. pp. 38–57.
4. Dezani-ciancaglini, M., Yoshida, N.: Asynchronous Session Types and Progress for Object-oriented Languages. In: In FMOODS'07. pp. 1–31.
5. Fähndrich, M., Aiken, M., Hawblitzel, C., Hodson, O., Hunt, G., Larus, J.R., Levi, S.: Language Support for Fast and Reliable Message-based Communication in Singularity OS. In: EuroSys '06. pp. 177–190.
6. Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley. (2002)
7. Gay, S., Vasconcelos, V., Ravara, A.: Session Types for Inter-process Communication. Tech. rep., University of Glasgow (2003)
8. Gay, S., Vasconcelos, V., Ravara, A., Gesbert, N., Caldeira, A.: Modular Session Types for Distributed Object-oriented Programming. In: POPL 2010.
9. Honda, K., Vasconcelos, V.T., Kubo, M.: Language Primitives and Type Discipline for Structured Communication-Based Programming. In: ESOP'98. pp, 122–138
10. Honda, K., Yoshida, N., Carbone, M.: Multiparty Asynchronous Session Types. In: POPL 2008. pp. 273–284.
11. Hu, R., Yoshida, N., Honda, K.: Session-Based Distributed Programming in Java. In: ECOOP 2008. pp. 516–541.
12. Ibrahim, A., Jiao, Y., Tilevich, E., Cook, W.R.: Remote Batch Invocation for Compositional Object Services. In: ECOOP 2009. pp. 595–617.
13. Song, Y.J., Aguilera, M.K., Kotla, R., Malkhi, D.: Rpc Chains: Efficient Client-Server Communication in Geodistributed Systems. In: NSDI 2009. pp. 17–30.
14. Takeuchi, K., Honda, K., Kubo, M.: An Interaction-based Language and its Typing System. In: In PARLE'94. pp. 398–413.
15. Vallecillo, A., Vasconcelos, V.T., Ravara, A.: Typing the behavior of Software Components using Session Types. Fundam. Inf. 73(4), 583–598 (2006)
16. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An Integrated Experimental Environment for Distributed Systems and Networks. In: NSDI 2002. pp. 255–270.
17. Yeung, K.C., Kelly, P.H.J.: Optimising Java RMI Programs by Communication Restructuring. In: Middleware '03. pp. 324–343.