Efficient Sessions

K.C. Sivaramakrishnan, Mohammad Qudeisat, Lukasz Ziarek, Karthik Nagaraj, Patrick Eugster

Purdue University, West Lafayette, IN, USA

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 a 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 for performance improvements. In this paper, we present a session type guided performance enhancement framework for distributed object interaction in Java. Our framework combines control flow information from MPSTs with data flow information obtained from corresponding programs. Detailed experimental evaluation of our distributed runtime infrastructure in both Emulab and Amazon's Elastic Compute Cloud (EC2) illustrate benefits of our composable enhancement strategies.

1. Introduction

Interaction between software components is one of the fundamental concerns of software development, yet creating a correct and efficient implementation remains a difficult endeavor.

1.1. Implementing Distributed Interaction

Real-world distributed systems involve multiple remote components, independently communicating via messages and coordinating activities among one another. Such interactions are often 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. Web services, e-commerce applications, and protocols like SMTP,

Email addresses: chandras@cs.purdue.edu (K.C. Sivaramakrishnan), mqudeisa@cs.purdue.edu (Mohammad Qudeisat), lziarek@cs.purdue.edu (Lukasz Ziarek), knagara@cs.purdue.edu (Karthik Nagaraj), peugster@cs.purdue.edu (Patrick Eugster)

POP3, are just some examples for structured protocols involving multiple interacting parties.

1.2. RPC et al.

Derivatives of the *remote procedure call* (RPC) abstraction [1] are (still) widely used for building distributed object systems due to their ability to provide some guarantees at least on typing of *individual* interactions. Examples include Java's inherent *remote method invocation* (RMI) [2] paradigm or the abstraction underlying the Common Object Request Broker Architecture (CORBA) [3]. These systems hide the details of network communication, allowing a programmer to simply invoke methods on remote objects as if they were local, ensuring type safety and supporting polymorphism on RMI calls. Unfortunately, this form of distributed object programming is inherently limited to expressing simple client/server interaction patterns and are plagued by a variety of overheads. In particular, multiple consecutive or follow-up RMI calls on a same object are treated entirely independently, each requiring a round-trip. In a high latency network, the client would spend most of its time stalled on responses from the server. In this manner, excessive RMI calls quickly lead to serious scalability and performance concerns. Object-oriented design advocates extensibility through the use of fine grained getters and setters to manipulate objects. However, such a pattern is ill-suited for a distributed environment, since every remote method call involves a network round-trip.

1.3. Existing Performance Improvements

A standard technique to overcome the network delays is to structure code based on *data transfer objects* (DTOs) or *remote facade* patterns [4]. DTO is a design pattern where a new object is defined just for the purpose of being transferred to a remote location. Typically, values are assigned to a DTO through the constructor and the object provides getters to fetch the values. Remote facade provides a coarse grained facade over fine-grained objects for efficient access over a network. Essentially, instead of making multiple remote calls, the aim is to combine the intent into a fewer coarse-grained remote calls. Such patterns advocate that new remote interfaces be defined in the `Member` class specifically for each objective. Although such a specialized definition can reduce the number of RMI calls, 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; RMI enhancement strategies have been especially well studied [5]. For instance, *batching* a contiguous subset of RMI calls together can be used to reduce the number of network round-trips [6]; instead of performing individual RMI calls one-by-one and waiting on each of their returns, the batching runtime at the client side creates a pipeline of RMI calls that need to be executed on the server. The runtime facilitates block transfer of method arguments and results. Batching is beneficial in a typical high latency environment like the

```
void inviteCoworkers(Member me, Date date) {
  Event evt = me.createEvent("Party", date);
  Employer myEmp = me.getEmployer();
  Location myLoc = me.getLocation();
  for (Member mbr : me.getFriends()) {
    if (myEmp.equals(mbr.getEmployer())
        && myLoc.equals(mbr.getLocation()))
      mailSvr.sendMail(mbr.getEmailAddress(), evt);
  }
}
```

Figure 1: Client implementation to invite co-workers to a party

Internet, where the round-trip time (RTT) dominates the transmission delay of the batch. RPC *chaining* [7] proposes composition of multiple consecutive remote calls to different servers, into a single chain of messages to save network latency from multiple round-trips. These enhancement strategies thus far are limited to bi-party interaction, semantically restricted to part of the protocol [6], or require code to be written in a style amenable to enhancement [7]. Earlier work by Mostrous et al. [8, 9] has studied the correctness of communication optimizations in the presence of code mobility through asynchronous subtyping. While the work focuses on correctness of optimizations, our work illustrates how session types themselves can exhibit opportunities for enhancements directly by exposing the distributed control flow behavior of the program.

1.4. Session Types

Recall that RPC and derived paradigms only allow for individual exchanges to be verified but not their "composition". *Session types* have been proposed as a way to precisely capture *complex* interactions between peers [10]. 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 and verifying interaction limited to two parties. They have since been extended to *multi-party session types* (MPSTs) [11] and objects [12]. Consider the example in Figure 1 which describes a simple protocol, implemented in Java RMI for simplicity, to invite co-workers (obtained from a social networking database) 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. Figure 2 shows the communication patterns that the client, information server, and mail server engage in.

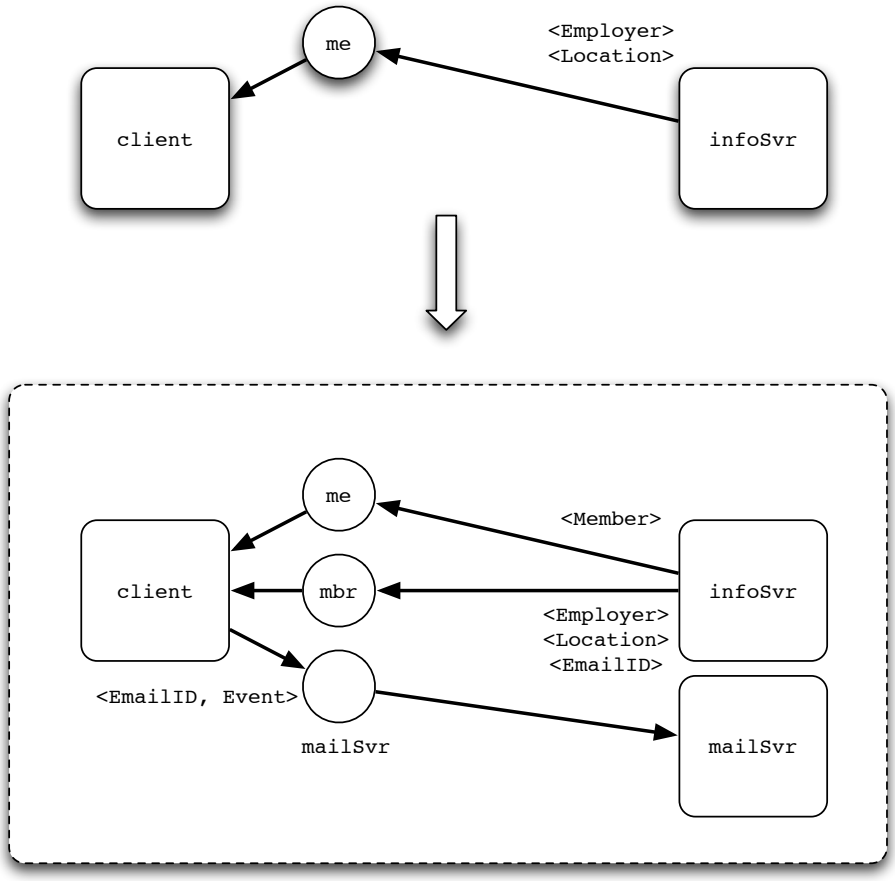


Figure 2: Communication pattern for the code presented in Figure 1. Squares represent communicating parties, circles show remote objects, and arrows depict communications. The dotted box represents communications done in a loop.

```
protocol invitation {
  participants: client, infoSvr, mailSvr
  .infoSvr->client: <Employer>
  .infoSvr->client: <Location>
  .infoSvr:
    [infoSvr->client: <Member>
     .infoSvr->client: <Employer>
     .infoSvr->client: <Location>
     .infoSvr->client: <EmailID>
     .client: {
       INVITE: client->mailSvr: <EmailID>
               .client->mailSvr: <Event>,
       NOOP:
     }
    ]*
}
```

Figure 3: Global session type of invitation example

Figure 3 illustrates the abstract description of the invitation protocol for the example described in Figure 1. The programmer defines a new global session type [11] through the use of a **`protocol`** block. This block explicitly defines (i) the participants of the protocol, (ii) the types of messages exchanged between the participants, and (iii) the order in which the messages are exchanged. Messages are sent asynchronously between the participants. 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, `infoSvr:[...]*` (see lines 6 to 15). `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 14). 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 to the protocol.

1.5. Session Type Guided Interaction

We observe that explicit control flow information manifest in MPSTs opens intriguing avenues for global performance enhancement of distributed multiparty interaction. In this paper, we present a session type guided framework for enhancing performance of distributed Java object interaction. Our approach however does not stop at individual, previously described bi-party enhancements and the use of MPST information solely. We introduce *combined* type- and compiler driven performance enhancements extending previously described bi-

party enhancements to multiple participants and allowing for their seamless composition.

Consider the invitation protocol example. It is evident from the corresponding session type that the first two messages from the `infoSvr`, namely `Employer` and `Location`, can be batched together as a single message. Similarly, the first four messages in the recursive type can be 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 define all remote interactions, we can statically assert that there is no causal dependence between a message sent to `mailSvr` and subsequent iteration decisions of `infoSvr`. With this knowledge, the code can be rewritten such that information about all of the friends 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.

1.6. Contributions

In this paper, we study the interaction, composability, and performance of enhancement strategies for distributed interacting objects. This is the first work that combines session types information with data flow analysis for composable enhancement of such interactions. We implement our enhancements through a framework called STING, including an optimizing compiler and distributed runtime infrastructure. The framework relies on an extension of Java, supporting specification of multi-party session types. Our MPST syntax extends the bi-party session type syntax from SessionJava [13]. In summary, the main contributions of the paper include:

- A Java extension that integrates multi-party session types.
- A detailed study of performance enhancements in the presence of a global interaction protocol expressed through MPSTs as well as a corresponding program.
- An empirical evaluation of enhancement strategies in a prototype framework, conducted in Emulab [14] and Amazon's Elastic Compute Cloud (EC2) [15].

1.7. Roadmap

The remainder of the paper is organized as follows. In Sec. 2, we present background information on session types. Sec. 3 describes our performance enhancements derived from session types and program analysis. The design of our compiler and runtime system is presented in Sec. 4. A detailed study of performance benefits through enhancements is given in Sec. 5 and Sec. 6 discusses their implications. Related work is described in Sec. 7. Sec. 8 concludes with final remarks.

2. Programming with Session Types

This section recalls concepts underlying session types and presents the syntax and semantics of our Java implementation of MPSTs which is inspired by the implementation of bi-party session types of Hu et al. [13].

2.1. Local Types

While *global* session types [11] in a multi-party session provide a universal view of the protocol involving all of the participants, *local* session types reveal only those interactions for a given participant. A projection from a global session type to such a local session type for each participant is well-defined for a coherent global session type [11]. Figure 4 shows the local session types for the invitation protocol projected from the global session types in Figure 3. 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.

As in related systems, the programmer implements only the global session types. The compiler automatically generates the local session types and checks the session implementation for conformance. In particular, the local session types presented in the paper are just internal representations of the compiler, and are not part of the session type syntax exposed to the programmer.

The local type for the `infoSvr` is given in **`protocol`** `invitation@infoSvr`. 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, the 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.

2.2. Session Initiation

Figure 6 shows how a new session is initiated and how other participants join the session with the help of `SJService`. A participant can participate in a protocol by using the `create` method, and providing the protocol and participant name. Multiple instances of the protocol can be executed concurrently by instantiating new `SJService` objects. Other participants can be added to the protocol using the `addParticipant` method and providing the participant location through a host name and port number pair. The `request` method initiates the protocol and returns a *session socket group*. A session socket group allows a host to communicate with other participants as dictated by the respective protocol. Communications on the session socket group are type checked using the corresponding local type to ensure that the participant adheres to the protocol.

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

protocol invitation@client {
 .infoSvr: ?<Employer>
 .infoSvr: ?<Location>
 .infoSvr:
    ?[infoSvr: ?<Member>
      .infoSvr: ?<Employer>
      .infoSvr: ?<Location>
      .infoSvr: ?<EmailID>
      !{INVITE: mailSvr:!<EmailID>
                .mailSvr:!<Event>,
        NOOP:
       }
      ]*
}

protocol invitation@infoSvr {
 .client: !<Employer>
 .client: !<Location>
 .![client: !<Member>
   .client: !<Employer>
   .client: !<Location>
   .client: !<EmailID>
   .client:
      ?{INVITE:, NOOP:}
  ]*
}
```

Figure 4: Local session types for `mailSvr`, `infoSvr` and `client`

2.3. Session Implementation

Once a session socket group has been created, the programmer uses a special application programming interface (API) for sessions — the *session API* — to implement the actual communication and control structures. Figure 5 shows the mapping between protocol description syntax and session API, which extends the one proposed by Hu et al. [13] for bi-party session types. 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 *session channels*, possibly shared, for communication between peers [11]. We assume that each participant has a unique channel to every other participant. We intentionally omit channel sharing to avoid proposing enhancements which could depend on such sharing. This also avoids conflicts with linearity analysis. Inversely, the implicit naming of participants by designating channels statically exposes fine-grained information on interaction. By similar lines of reasoning we do not take delegation into account and, as in most work on session types, we assume that individual logical participants are single-threaded (multi-threading can allow for co-location of participants).

Protocol syntax	API call	Purpose
`A: !<T>`	`ss.send(obj, "A")`	sends `obj` of type `T`
`A: ?(T)`	`ss.receive("A")`	receives object of type `T`
`!{L:T, ...}`	`ss.outbranch(L){}`	body of case `L` of type `T`
`A: ?{L:T, ...}`	`ss.inbranch("A"){}`	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`

Figure 5: Protocol description syntax and its mapping to the session API

Using the session API described above, the client described in Figure 1 can be expressed as outlined in Figure 6. 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. Exceptions are 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 continue the protocol.

2.4. Extensions

Our multi-party session type implementation supports additional extensions to the session type description and the session API. These extensions capture the programmer's intent that enables our compiler and runtime system to precisely introduce optimizations when it is beneficial to do so, and provide fine-grained control over limiting the extent of optimizations.

2.4.1. Explicit buffer flushing

Batching can be undesirable in situations where the responsiveness of the application is a priority. An example scenario would be an `ssh` session, where an undue delay in the response is detrimental to the utility of the application. Since our enhancements are not aware of the quality of service requirements of the application, our enhancements can be problematic in certain instances. Hence,

```
void inviteCoworkers() {
  final noalias SJService service =
    SJService.create(invitation, "client");
  service.addParticipant("infoSvr", "infoSvr.host", 8888);
  service.addParticipant("mailSvr", "mailSvr.host", 9999);
  final noalias SJSocketGroup ss = service.request();
  Employer myEmp = (Employer)ss.receive("infoSvr");
  Location myLoc = (Location)ss.receive("infoSvr");
  Event evt = me.createEvent("Movie", date);
  ss.inwhile("infoSvr") {
    Member m = (Member)ss.receive("infoSvr");
    Employer emp = (Employer)ss.receive("infoSvr");
    Location loc = (Location)ss.receive("infoSvr");
    EmailID eid = (EmailID)ss.receive("infoSvr");
    if (myEmp.equals(emp) && myLoc.equals(loc)) {
      ss.outbranch(INVITE) {
        ss.send(eid, "mailSvr");
        ss.send(evt, "mailSvr");
      }
    } else {
      ss.outbranch(NOOP) {}
    }
  }
}
```

Figure 6: Client implementation of invitation protocol using session API

our system provides a way to manually flush send buffers, using the `flushBuf` method on the session socket group. By invoking this method, programmers can control the limit for batching. Although we envision our optimizations to be applied for applications where this kind of responsiveness is not a priority, the `flushBuf` method can be used to selectively specify instances where sends need to be discharged without delay.

2.4.2. Deployment characteristics

The information about the available bandwidth and inter-network latencies between the participants can be supplied as an optional parameter along with the global protocol description. Although this information can be measured *after* deployment, it becomes necessary to supply this information explicitly, if the compiler is to make informed optimization decisions, which happens *before* deployment. This information is utilized, for example, to decide that batching might not be profitable if the latency between the participants is very less.

Optimization	Type inspector	Compiler	Runtime
Batching sends	Identify opportunities	Transform	Compose & tune
Choice lifting	Identify opportunities	Transform	Compose & tune
Batching sends with DFA	–	Identify opportunities & transform	Compose & tune
Chaining	Identify opportunities	Transform	–
Continuation exporting	Identify opportunities (through annotations)	Transform	–

Table 1: Summary of tasks involved in different phases of the system with respect to optimizations. Composability refers to the composition of optimization opportunities identified statically and dynamically by the infrastructure.

2.4.3. Performance characteristics

Session types do not reveal whether a particular participant is a thin, mobile client, with limited computing resources, or a fat server, with ample processing power. This information is necessary to decide whether certain optimizations that migrate computations (Sec. 3.5) do not introduce detrimental effects. To this end, we introduce an optional parameter in the global protocol description, to capture the performance characteristics of participants. The participants in the global protocol can be optionally tagged as belonging to one of the three classes – `SERVER`, `CLIENT`, and `MOBILE`, presented in the decreasing order of computational power. Computation, if migrated, is always from a participant with less computing power to one with more.

3. Performance Enhancement Strategies

This section presents our session type guided performance enhancements. In general, the optimization opportunities are inferred from the session types, which are passed to the optimizing compiler. The session compiler performs modular code transformations that express the optimizations, while preserving the protocol correctness. The session runtime performs further optimizations by composing together optimizations performed by the compiler. The session runtime also tunes the optimizations, such as limiting the size of batches, performing timed flushes, etc. Table 1 summarizes the tasks performed in each phase of the system with respect to the optimizations. In this section, we will present the enhancements in the increasing order of complexity along with the explanation for tasks performed in each phase of the system.

3.1. 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 Figure 3, the two sends at the beginning are tagged as batchable by the type inspector. The compiler then performs code transformation to remove the implicit flush at the end of the first message send, such that the messages are not flushed immediately. This provides opportunity for the runtime to wait for further messages to construct a batch. These batched sends are represented in the *enhanced session type* as:

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

The enhanced session types are only used as a way to illustrate our optimizations and are not a programming idiom. As illustrated in Table. 1, the type inspector identifies the batching opportunities, which is utilized by the compiler and runtime system to perform the optimization. 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 client.

The runtime at the client decouples the messages and passes them individually to the program. Batching, therefore, remains transparent to the program. The type inspector also exposes opportunities for batching sends in a recursive type if the participant acting as the loop guard does not perform any receives in the body of the loop. The following example shows a global session type, that captures the interaction between three participants; `server`, `client1` and `client2`. In every iteration of the outwhile loop, the `server` sends to `client1` a string followed by `client1` sending to `client2` an **`int`**. Also, the server decides if the next iteration of the loop has to be executed. This information is broadcast to other participants, which also execute an iteration, if the server decides to do so.

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

is enhanced 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 enhanced 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 across multiple iterations of 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 the following global session type:

```
server->client: <bool>
.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.

3.2. 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>}]*
```

Apart from the two application generated messages, the runtime also sends two control messages in each iteration. Both control messages are sent from `A` to `B`. The first control message is for the recursion informing `B` to execute one more iteration, and the second message is the branch label chosen by `A`. Thus, the number of messages sent in the above protocol is $1 + 4 \times num\_iterations$.

The type inspector identifies that 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) and any local decisions made at `A`, the choice can be lifted as far as the most recent message reception. The compiler performs code transformation, the result of which is captured by the following enhanced session type:

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

Participant `B` first sends a boolean message to `A`. This is followed by a single batched message from `A` to `B`, where each individual message inside the batch is `<L1, int, String>` or `<L2, int, bool>`. In other words, each message has the label information (*control messages*) batched along with the payload. Since there are no intervening receives in `A`, all such messages can be batched. The enhanced session type needs to perform just two message sends. If the optimized choice were to be followed by another message send or a series of sends from `A` to `B`, our runtime will compose together the optimized choice with the subsequent batch into a single message. Thus, batching sends are composable with choice lifting as mentioned in Table. 1.

3.3. Batching sends with 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 the item. Consider the modified version of the invitation example presented in Figure 3, where there is a single participant named `server` instead of `infoSvr` and `mailSvr`. As with the original example, based on the employer and location of the member, the client chooses to invite the member to the party. We define the local session type at server below.

```
protocol invitation@server {
  .client: !<Employer>
  .client: !<Location>
  .![client: !<Member>
    .client: !<Employer>
    .client: !<Location>
    .client: !<EmailID>
    .client:
      ?{INVITE: client: ?<EmailID>
                .client: ?<Event>,
        NOOP:
       }
   ]*
}
```

The following snippet shows the enhanced version of this type, where consecutive sends have been batched through our batching optimization defined in Section 3.1.

```
protocol invitation@server {
  .client: !<Employer, Location>
  .![client: !<Member, Employer, Location, EmailID>
    .client:
      ?{INVITE: client: ?<EmailID, Event>,
        NOOP:
       }
   ]*
}
```

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 two values of type `EmailID` and `Event` corresponding to `INVITE` branch, or receive no values corresponding to `NOOP` branch. 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 give us the control flow information, but no data flow information. Hence, our type inspector will not be able to identify further batching opportunities.

To address this limitation, we implement a 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 [6]. Our analysis is an extension of standard "def-use" analysis, and transitively tracks any received value that might flow to a send or affect the looping decision on an outwhile. If independence of a "use" cannot be asserted, we pessimistically avoid such batching to retain safety.

Session type information allows us to precisely determine the scope of the analysis. In the above example, the data flow analysis might show that neither the loop conditional nor the sends are dependent on the receives. In this case, we can enhance the session type as below, in which all of the sends are batched, followed by a second batch with all receives.

```
protocol invitation@server {
  .client: !<Employer, Location>
  .client: !<Member, Employer, Location, EmailID>*
  .client:
    ?{INVITE: client: ?<EmailID, Event>,
      NOOP:
     }*
}
```

Again, note that the runtime can perform further optimizations by composing the optimized loop with subsequent messages, if any, from the client to the server.

3.4. Chaining

Chaining is a technique to reduce the number of cross-site RMI calls. Chaining can significantly reduce end-to-end latency in a setting where the participants are geo-distributed [7]. Chaining helps to avoid forwarding patterns in the protocol. Our implementation leverages continuations to implement classic chaining optimizations. To illustrate, consider a user shopping for music from his phone in an online music store like Rhapsody, Amazon MP3, etc., over a 3G network. The user requests an album and buys songs with ratings higher than 8 of 10 from that album. The user then transfers the songs to his/her desktop personal computer (PC), which is connected to the Internet. The following snippet shows the pseudo-code for the phone written in an RMI style. Figure 7 graphically depicts the communication protocol for the code given below.

```
void onlineShopping() {
  Album a = Vendor.album ("Electric␣Ladyland");
  for (SongInfo si : a) {
    if (si.rating() > 8) {
      Song s = si.buy();
      PC.put(s);
    }
  }
}
```

The corresponding session type for this protocol is given hereafter.

```
protocol onlineShopping {
  participants: vendor, phone, PC
  .phone->vendor: <String>
  .vendor->phone: <Album>
  .vendor:
  [vendor->phone: <SongInfo>
   .vendor->phone: <int>
   .phone: {BUY: vendor->phone: <Song>
                 .phone->PC: <Song>,
            NOOP:
           }
  ]*
}
```

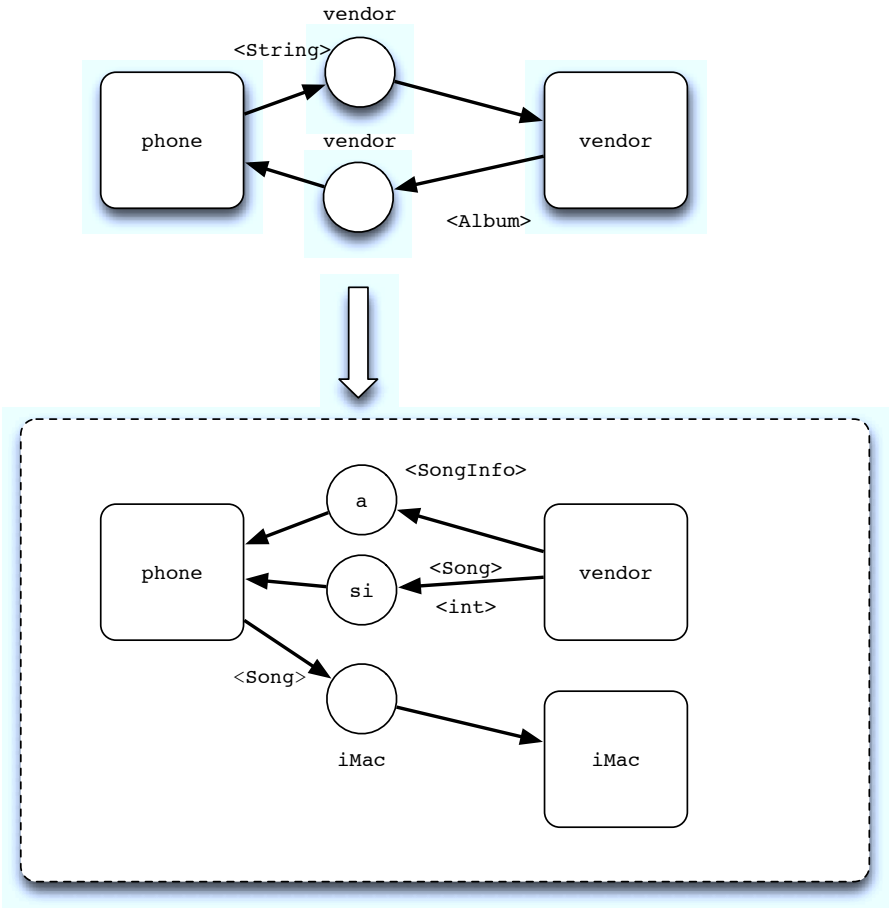


Figure 7: Communication pattern for the `onlineShopping` method. Squares represent communicating parties, circles show remote objects, and arrows depict communications. The dotted box represents communications done in a loop.

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

```
!{BUY: vendor: ?(Song) .PC: !<Song>, NOOP: }
```

The type inspector observes that the phone receives 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. Our implementation leverages continuation exporting to implement classic chaining. 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. The optimized global session type for the online shopping example is shown below.

```
protocol onlineShopping {
  participants: vendor, phoneExp, PC
  phoneExp->vendor: <String>
  .vendor->phoneExp: <Album>
  .vendor->phoneExp: <SongInfo, int>*
  .phoneExp->vendor: {BUY|NOOP}*
  .vendor->phoneExp: <Song>*
  .phoneExp->PC: <Song>*
}
```

The new participant `phoneExp` is the exported version of phone, running on the same host as the vendor. Hence, the communication between vendor and `phoneExp` is local. As soon as the protocol is started, the continuation from the phone is exported to the site of vendor. The vendor and `phoneExp` exchange local messages to determine the songs to be transmitted, and finally the songs are transmitted to the PC in one batch.

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).

3.5. Exporting continuations

The continuation exporting leveraged to implement classic chaining can be extended for more general optimizations. Let us reexamine the example in Figure 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. Our 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 can execute the 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 the 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 code with the readily available client code. Since our enhancements are performed statically, we are able to compile the remote code into the local code. This is a similar approach adopted by multi-tier programming [16, 17, 18].

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 Figure 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 if applied without care, 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 benefit thin clients and fat servers (see Sec. 5.3).

4. System Design

This section describes the design of our system and runtime architecture used for implementing the enhancements presented previously.

4.1. Overview

Figure 8 depicts an overview of the process. The STING compiler is a combination of a type inspector and Java compiler, whereas the STING runtime is a veneer on top of the Java virtual machine (JVM). The compiler is utilized to

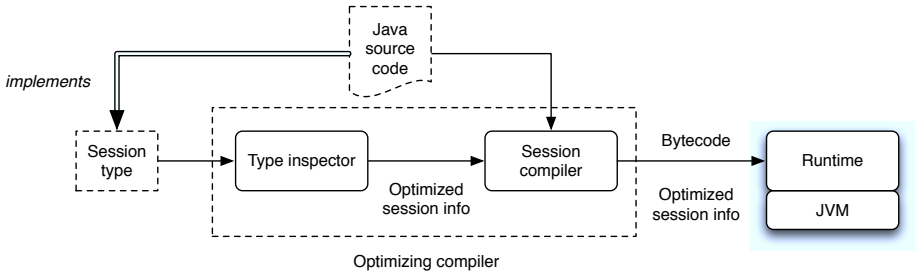


Figure 8: System design with enhancement strategies.

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 enhancements. Decisions to utilize these enhancements are made at runtime. The STING compiler has been implemented using Polyglot [19] compiler framework. We utilize Javaflow continuation library [20] for capturing exportable continuations and resuming them at a later point in time.

4.2. STING *compiler*

The main tasks of the STING compiler include verifying the conformance of participant implementation to the session types, analyzing and performing various enhancements, and generating runtime batching rules for the STING runtime. We will look at some of the interesting implementation details below.

4.2.1. Type checker

The STING compiler implements a session type verifier that first checks if the protocol description is well-formed. Then it generates the local session types for each participant and verifies the conformance of an individual participant to its corresponding local session type. A session socket group can be passed as an argument to a function, in which case, the local type of the remainder of the protocol is the type of the argument. Subtyping is allowed on message types, which enables an instance of a subclass of a prescribed class to be sent. The type checker verifies that the string literal in `send` and `receive` commands in the session API corresponds to a valid participant in the protocol.

4.2.2. Translation

The compiler translates the session API to downcalls to the STING runtime API. The `send` call on the STING runtime API just buffers the messages; when the compiler encounters a reception from a participant, it inserts a flush command on the corresponding participant's sending buffer. This forces the runtime to package the messages and send them to the corresponding receiver.

The compiler also translates implicit control flow information in the session API to explicit messages. `outbranch` and `outwhile` are translated to multisends to disseminate the control flow information to all participants. `inbranch`

is translated to a `receive` followed by a **`switch`** statement on the branches, and `inwhile` is translated to receiving a boolean in the loop conditional. The compiler performs code reorganization for exporting continuations based on the result of static analysis.

4.3. Sting *Runtime*

Our runtime system is designed as a veneer between a JVM and the user code that conforms to the session types. The runtime exposes a set of APIs that is used for exchanging the messages defined by session types. This API provides the methods defined in Sec. 2.3. The runtime uses TCP sockets to establish connections with other participants and exchange network messages.

4.3.1. Intermediate representations

The Sting runtime stores an intermediate representation of the optimized session types for use in online decisions. As the runtime gets requests for operations, it consults session type representations to infer the best action to perform. All batching or chaining decisions are made at the initiator runtime. Since this controls the future consequences of the messages, the decisions will always be consistent with the corresponding receivers.

We construct a deterministic finite state machine from the local session type information during initiation. The transitions represent the `send` and the `recv` events, and the states represent some valid state of the protocol. The Sting compiler generates the state machine along with some special labeled transitions. These transitions identify situations at runtime when all queued messages should be batched and transmitted. For example, a transition that leads out of a loop iteration is generally an opportunity to batch all the loop sends to the receiver.

4.3.2. Session protocol messages

Whenever a `send` is called on the runtime, the payload is queued by the runtime for sending. If the runtime decides that the queued messages are to be sent to participant(s), it encodes queued sends into a single object to be transmitted on the network while preserving the order of sends. Each individual message sent is accompanied by meta-data including the receiver, type of the object, etc.

On the receiver side, the incoming messages are first processed by a *deoptimizer*, before reaching the application. The deoptimizer de-serializes the messages, and parses the batch and chain messages. Subsequently, the messages intended for the current participant are delivered to the application, and in the case of chain messages, the messages are forwarded to the intended recipient. Thus, the receive operation in the application is completely unaware of the sender-side optimizations. This design allows us to perform local optimizations on a particular participant during runtime, without affecting the intended communication behavior on the matching participant.

On the sender side, the Sting runtime examines the state machine and flushes messages either through an explicit flush message if the next communication operation is a reception, or if the message is the last message in the

protocol. Otherwise the runtime waits for the number of messages buffered to reach a flush threshold, at which point the messages are serialized and flushed. The runtime also flushes out the messages periodically. The period is tunable by the application.

4.3.3. Chaining

When sends are required to be chained along a set of servers, the runtime encodes the intended recipient of each of the messages into the chain message. When a particular receiver in the chain receives the chain, in extracts the messages meant for it and enqueues it to the corresponding local channel. Then, it forwards the rest of the chain to the next receiver along the chain. The next participant is usually the next receiver in the message ordering, but a better decision could possibly be made in some scenarios based on network topology. The original sender of the chain is suspended till all of the messages in the chain have been enqueued on the appropriate channel.

5. Evaluation

We evaluate the performance gains of our enhancements from experiments that characterize typical scenarios in distributed systems. In particular, we evaluate the system in a simulated environment in Emulab [14] as well as in the Amazon Elastic Compute Cloud (EC2) [15]. Recently, EC2 has become a popular choice of platform for scalable deployment of distributed systems, and this evaluation highlights the substantial benefits from these enhancements.

5.1. Experimental Setup

We first describe the experimental setup for each of these scenarios. Sun Java 1.6 was used for running all our client and server Java programs. The results presented in each case are an average over 20 similar runs.

5.1.1. Emulab

Emulab is an environment for simulating a network of nodes and provides fine grained control over network characteristics such as latency and bandwidth. We used a two node setup for the batching experiments with 1 Mbps link bandwidth and round-trip times (RTTs) of 40, 80 and 150 ms, 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 1 Gb links with local area network latency interconnects. The Emulab machines had 850 MHz Pentium 3 processors with 512 MB RAM.

5.1.2. Amazon Elastic Compute Cloud (EC2)

Amazon EC2 allows users to rent virtual machine instances in the cloud. Amazon EC2 is becoming one of the most popular platforms for deploying scalable distributed applications. For our experiments, we deploy our applications

	US East	US West 1a	US West 1b	EU
US East	0	83	82	92
US West 1a	83	0	2	159
US West 1b	82	2	0	158
EU	92	159	158	0

Table 2: Round trip times (in ms) between Amazon EC2 datacenters

	US East	US West 1a	US West 1b	EU
US East	-	123	122	132
US West 1a	123	-	614	37
US West 1b	122	614	-	40
EU	132	37	40	-

Table 3: Bandwidth (in Mbps) between Amazon EC2 datacenters

on four geo-distributed datacenters, namely, US East, US West 1a, US West 1b and the European Union (EU). US West 1a and US West 1b are two co-located datacenters in the California area. We used *small* instances in each of the datacenters, which are allocated 1.7 GB of memory, 1 EC2 32-bit compute unit. The median RTT values between the datacenters are given in Tab. 2, and the bandwidths are given in Tab. 3. For our chaining experiments in EC2, we instantiate the servers in US West 1a, 1b which have a high bandwidth, and experiment with clients in other datacenters.

5.2. 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: <bool>
  ]*
```

This is implemented both using RPC style calls and within STING. For the RPC style implementation, signature sent from client to server is assumed to be the argument for the RPC, while the boolean value indicating the result of the test is analogous to the RPC result. 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.

Figure 9 shows the comparison of the successive sends and receives with batched sends in Emulab. We use an object size of 0.5 KB (since most objects

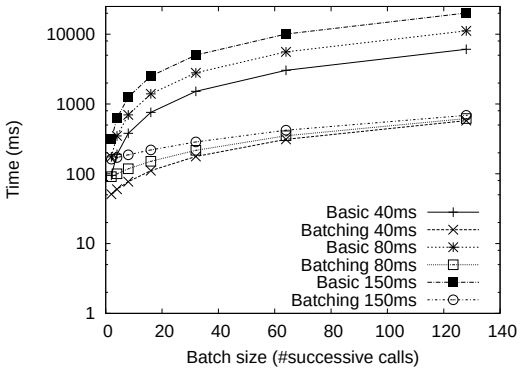


Figure 9: Performance of batched sends on Emulab

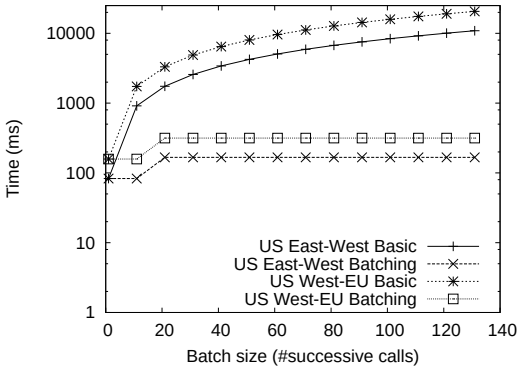


Figure 10: Performance of batched sends on Amazon EC2

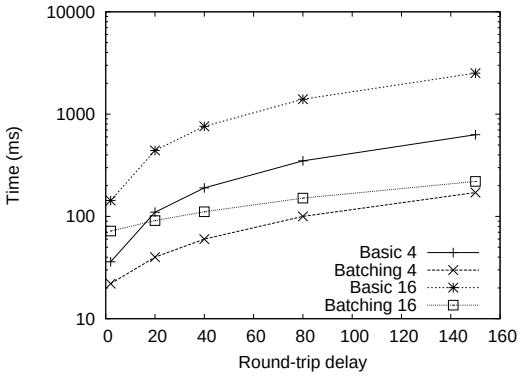
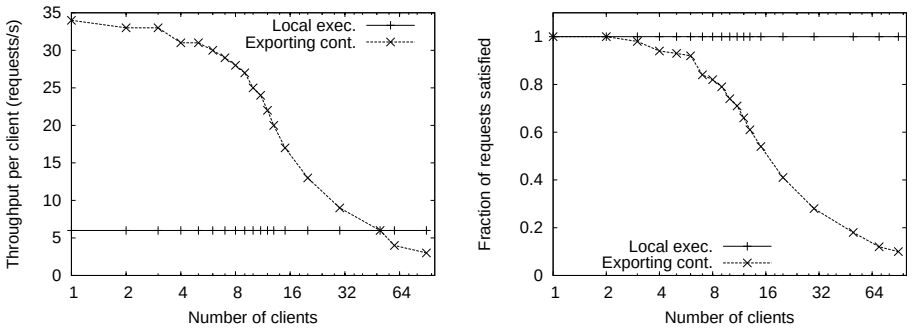


Figure 11: Batching performance with varying RTT in Emulab

in a typical Java program are small) for this experiment and compare the RTT values of 40, 80 and 150 ms. Note that the ordinate is in logarithmic scale. Our experiments show that the elapsed time is affected linearly with the batch size, and all the batched sends are faster than basic sends. An increase in RTT delay between the client and server increases the time per operation, thus widening the gap. This is because batching is only affected once by the RTT, but basic sends are affected on each instance. Thus batching has real benefits for large batch sizes and higher RTT delays.

We repeat the batching experiment described above in Amazon EC2. We evaluate benefits of batching for two pairs of datacenters; between US East and US West and between US West and EU. The results are shown in Figure 10. Even here, we see that as we increase the batch size, the total time taken by the basic case with successive sends increases linearly, whereas the total time for batching case remains constant except for the initial increase. We consistently and repeatedly noticed the doubling of latency when the packet size crossed a threshold at about 5 to 10 KB. This bump is not an artifact of our implementation as we do not observe this in the Emulab experiments. Previous analysis of vitalization in Amazon EC2 [21] has observed such inconsistencies and malformations in TCP and UDP network throughput. This does not affect the basic case as all those packets are just sized at 0.5 KB.

Figure 11 shows the effect of RTT on the cost of batching from experiments in Emulab. We vary the RTT in these experiments from 2 ms to 150 ms and measure the cost of normal and batched sends for objects of size 0.5 KB. We perform the experiments for 4 and 16 objects in one operation. As shown in the graph, increasing the RTT increases the time consumed for the sends. However for the normal sends, each of the sends is affected by one RTT and hence increases linearly with it. Batched sends however suffer from the cost of a single RTT alone and perform better. In the case of larger number of objects to be sent, the time taken increases in both cases, but the increase for batched



(a) Client (b) Server

Figure 12: Exporting continuation: throughput of requests in Emulab

sends is minimal.

As expected, the results showed that batching improves performance linearly with batch size. Batching performed better with increased network latency as the number of network round-trips were reduced.

5.3. 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 [22]. 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 3 GHz dual-core machine with 4 GB RAM. We run multiple instances of server program so as the satisfy multiple client requests. The stock data is placed in an in-memory database for efficient access. This reduces the influence of disk access latencies from affecting the results. The clients were executed on different machines with identical configuration.

5.3.1. Client throughput

Figure 12(a) shows the throughput of requests serviced per client as we increase the number of concurrent clients. The client requests are throttled at 34 requests/sec. 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

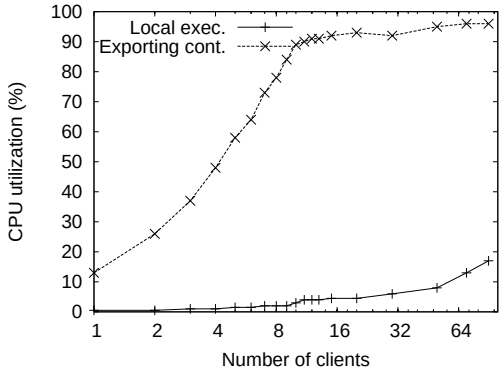


Figure 13: Exporting continuation: server CPU utilization in Emulab

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.

5.3.2. Server throughput

Figure 12(b) shows the fraction of requests satisfied per second. 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.

5.3.3. Server CPU utilization

Figure 13 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 the 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.

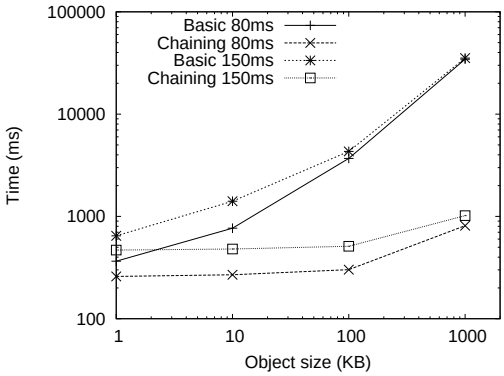


Figure 14: Benefits of chaining on Emulab

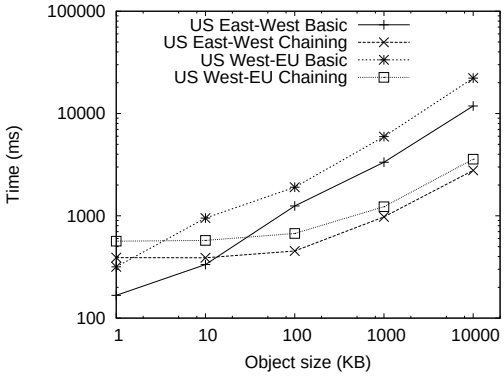


Figure 15: Benefits of chaining on Amazon EC2

Batching reduces the number of separate remote communication to 1 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 less 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.

5.4. Chaining

We implemented the example of purchasing songs from the phone discussed in Sec. 3.4, where chaining is exploited to reduce communication involving the slow phone. Figure 14 compared chained and non-chained versions in Emulab. For Emulab, we set the network up such that the phone has a 1 Mbps high latency link to the server and the PC, and the server is connected to the PC on

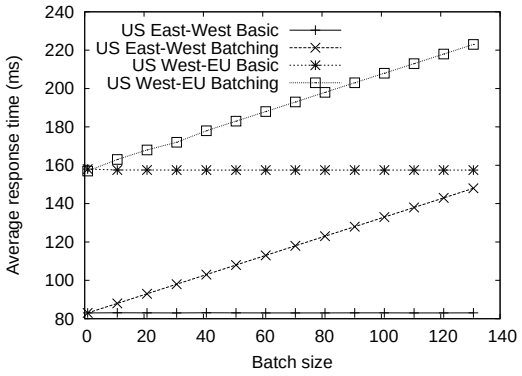


Figure 16: Effect of batch size on average response time in Amazon EC2

a 100 Mbps link with 2 ms latency. We vary the size of the object being chained from 0.1 KB to 1 MB and experiment with RTT values of 80 ms and 150 ms. The upper two curves in the graph show the basic approach (without chaining) used to transfer the object through the phone. In this case, the transfer speed 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. The basic case takes an order of magnitude more time to transfer a 1 MB song.

We repeated the above experiment in Amazon EC2. Since we do not have control over the latency or the network bandwidth, we create an instance each for the server and the PC in the co-located datacenters in the US West region. This simulates the low latency (2 ms), high bandwidth (614 Mbps) connectivity between the server and the PC. The phone instance is placed in a different datacenter in order to simulate the high latency network that the phone is attached, and we experiment with instances in US East and EU. The results are shown in Figure 15. We observe that the results are similar to the Emulab experiments with the chained version performing much better than the basic version due to the cost of transferring large files over a low bandwidth network. The benefits are lesser in this scenario because the bandwidth difference between the Phone and the servers in Amazon EC2 is an order of magnitude larger than in Emulab. Nevertheless, transferring a 1 MB song would complete in just about 15% of the time by leveraging chaining capabilities (US West - EU).

5.5. Determining Batch Size

In the batching experiments described in Sec. 5.2, we see that with increasing batch size, the total time taken to complete the protocol decreases. The flip-side to these advantages is the increase in response times for individual send-receive pairs. This is because the runtime does not eagerly perform the communication

action and waits for the batch to fill up before issuing the send. We ran the batching experiment and measured the average response time for each request as a function of the batch size in Amazon EC2. We simulate a 1 ms delay between successive requests to model time taken for local computations. The results are presented in Figure 16. In the basic case without batching, if the server can immediately satisfy the client request, the average response time is similar to the RTT between the server and the client. For US East-West case, this is 83 s. Since there is no batching involved, it is represented as a flat line. For the batched versions, the average response time increases linearly with increasing batch size. Considering a particular batch size, it is obvious that most of the difference is due to the delay enforced by the waiting for local computations between sends. This difference gets averaged on each of the send-receives.

If the application explicitly states maximum tolerable RTT as a quality of service parameter, the runtime can use this information to approximate the ideal batch size for a pair of participants. In the US East-West case, if the application can tolerate a network RTT of 120 ms, then the runtime chooses 75 as the batch size. Batches are also flushed after a maximum wait time as mentioned earlier, and real-time requests may be annotated to selectively avoid enhancements.

6. Discussion

Optimizing distributed system performance in a general manner is difficult. The performance of enhancements depends highly on interaction patterns, network characteristics, as well as code distribution and balancing. Our framework must address these issues in the enhancements it performs.

6.1. Assumptions

Though we have illustrated the latency gains offered by batching and chaining network packets, the underlying modifications change the computational overhead at each physical machine. As our experiments have illustrated, gathering more substantial computations at a single node on the network can have an adverse effect. Similarly, in applications described earlier for chaining messages through a set of nodes, it is assumed that the network links between the servers have larger bandwidth and lower latency (or equivalent) compared to links between the client and the servers. Although we could envision guiding our enhancement decisions by a specification of hosts and network configuration, like others [6, 7], we currently take decisions in a manner agnostic to the underlying infrastructure.

6.2. Extensions

Several ways exist to accommodate further enhancements through *additional syntax*. We mention two examples.

6.2.1. Guiding exportation

Our current implementation always exports code if it is possible. There are situations when the programmer knows that certain blocks of code should never be exported for enhancements. The server might want to prevent user authentication to be exported to the client, or choose to execute compute intensive code locally for better performance. Our system could be extended with a `fixLocation` block, providing the property that code within this block is never exported. To facilitate execution of remote code locally, we assume that during compilation, code from all of the participants is available. This precludes continuation exporting, if the participants were separately compiled. If we assume that participants are implemented as a collection of callback functions as described in [7], we can export continuations (callback functions) without the need for code from all participants.

6.2.2. Multicast

Our session type syntax could be further enriched with a specialized multicast primitive. A multicast primitive would allow a participant to send the same message to a collection of peers. This facilitates a type driven chaining enhancement where one copy of the message is published by the source to one participant, who then forwards the message to another participant in the multicast group, and so on. Such a primitive would be useful, especially in the case of sessions with a large number of participants.

6.3. Failures

Shifting load among nodes in a distributed system can also affect fault tolerance, as individual nodes might end up doing more and different work than originally foreseen. Overall, however, any session can fail if a single involved process fails. In this sense, our enhancements do not change the global characteristics. Further fault tolerance mechanisms could be added, possibly with further syntax in session types, to also enhance distributed object interaction protocols in terms of their fault tolerance.

6.4. Cloud Environment

Cloud environments are the new shift toward using a small share of resources from a datacenter simply by paying for their usage (Amazon EC2 [15], Microsoft Azure [23], etc.). Companies can now rent servers and pay per their use instead of having to purchase and manage server quality hardware connected to the Internet. Cloud environments typically provide elasticity of resources, by which more servers can be added to the processing path for faster response times. Elasticity is primarily leveraged using a load balancer, which balances the requests onto the server hardware. Another method of elastic computing includes distribution of computation among a group of servers based on the number of servers in the pool (Google Cluster Architecture [24]). Such situations warrant the need for collective collaboration among the servers for executing the task.

Session types abstract away from the actual physical hosts on which they are executed. Hence a participant defined in the session type is not tied to a single host. A load balancer can be used to pick any host to forward a request thus achieving elasticity, without the need for modifying the session type. Some similar operations that are transparent to the types such as replication and fail-over to different hosts can also be made possible. However, in some cases Sting might be able to leverage the network setup to provide performance benefits, but session types do not have provision to specify a collection of servers. Such an extension would allow for fine-grained enhancements for the cloud environment.

The cloud is a *controlled* environment in terms of observed network bandwidths and latencies, because of high speed switching facilities within datacenters and backbone links on the Internet. It is easy to formulate these guarantees into specifications for Sting, to produce valid enhancements.

7. Related Work

A number of performance enhancements for distributed object systems have been proposed. We focus here on the efforts closest related to session types and our proposed enhancements.

7.1. Session Types

Session types [10] allow precise specification of typed distributed interaction. Neubauer and Thiemann first described the operational semantics for *asynchronous* session types [25]. Early session types described interaction between two parties, which has then been extended to multi-party interaction by Honda et al. [11] and Bonelli et al. [26]. 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 [27] extends that of Honda et al. [11] 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 been applied to functional [28], component-based [29], and object-oriented settings [30, 13]. Asynchronous session types have been studied for Java [31]. Bi-party session types have been implemented in Java [13]. Our protocol description syntax is inspired from the syntax described in that latter work. Our framework supports multi-party protocol specification and interaction. Gay et al. [12] 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. Scribble [32] is an ongoing project to develop a session type based formal protocol description language and a tool chain to help build large scale distributed applications. Since the distributed participants adhere to the communication protocol, we believe our insights into global session based optimizations are applicable for protocol enhancement strategies in Scribble.

7.2. Batching

The remote procedure call (RPC) paradigm has been criticized for its coupling between distributed participants and lack of composability [33]. Explicit batching through the use of remote facade pattern and data transfer objects has been proposed as an alternative to fine-grained remote procedure calls [4]. But this requires programmers to implement specialized methods for each client access pattern. Batched *futures* [34] mask the latency of remote procedure calls by having the result of the invocation sent asynchronously back to the caller. Yeung and Kelly [35] 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) [6] 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 cannot batch a loop, which requires user input on every iteration. RBI is also limited to batching specialized control structures and is unable to handle arbitrary computation. Our system allows global optimization decisions to be made, and can batch across *multiple* participants. With the help of session types, we can perform type driven control flow optimizations like loop flattening and choice lifting, which are not supported by other systems.

7.3. Chaining

First-class continuations [36] are a general idea to allow arbitrary computation to be captured and sent as arguments to other functions. In a distributed setting, exporting continuations is advantageous where the cost of moving data is much larger than the cost of moving computation. *RPC chains* [7] reduce cross site remote procedure call overheads by exporting callback functions to the remote host. This system requires that the user writes code in a non-intuitive *continuation passing* style [36]. Another limitation is that 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 continuation passing style.

8. 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 program analysis can yield further performance enhancements for distributed object interaction. We have demonstrated the benefits of our approach in two network settings – Emulab and Amazon's Elastic Compute Cloud (EC2). In particular, we have shown that our continuation exportation scheme benefits applications with thin clients and fat servers.

We are in the process of investigating interaction between our enhancements and advanced session features like delegation, channel sharing, and participants assuming multiple roles or using multiple threads.

Acknowledgments

We are very grateful to A. Ibrahim, Y. Jiao, E. Tilevich, and W. Cook for sharing insights and the source code for RBI [6] with us.

References

[1] A. Birrell, B. Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems 2 (1) (1984) 39–59.

[2] Sun, Java Remote Method Invocation - Distributed Computing for Java (White Paper) (1999).

[3] OMG, The Common Object Request Broker: Architecture and Specification, OMG, 2001.

[4] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., 2002.

[5] J. Maassen, R. V. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for Parallel Programming, ACM Transactions on Programing Languages and Systems 23 (6) (2001) pp. 747–775.

[6] A. Ibrahim, Y. Jiao, E. Tilevich, W. R. Cook, Remote Batch Invocation for Compositional Object Services, in: Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009), 2009, pp. 595–617.

[7] Y. J. Song, M. K. Aguilera, R. Kotla, D. Malkhi, RPC Chains: Efficient Client-Server Communication in Geodistributed Systems, in: Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09), 2009, pp. 17–30.

[8] D. Mostrous, N. Yoshida, Session-based communication optimisation for higher-order mobile processes, in: Proceedings of the 10th International Conference on Typed Lambda Calculus and Applications (TLCA 2009), 2009, pp. 203–218.

[9] D. Mostrous, N. Yoshida, K. Honda, Global principal typing in partially commutative asynchronous sessions, in: Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP 2009), 2009, pp. 316–332.

[10] K. Takeuchi, K. Honda, M. Kubo, An Interaction-based Language and its Typing System, in: Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe (PARLE '94), 1994, pp. 398–413.

[11] K. Honda, N. Yoshida, M. Carbone, Multiparty Asynchronous Session Types, in: Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL '08), 2008, pp. 273–284.

[12] S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, A. Z. Caldeira, Modular Session Types for Distributed Object-Oriented Programming, in: Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL '10), 2010, pp. 299–312.

[13] R. Hu, N. Yoshida, K. Honda, Session-Based Distributed Programming in Java, in: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP 2008), 2008, pp. 516–541.

[14] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar, An Integrated Experimental Environment for Distributed Systems and Networks, in: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02), 2002, pp. 255–270.

[15] Amazon EC2.
URL `http://aws.amazon.com/ec2/`

[16] M. Neubauer, P. Thiemann, From sequential programs to multi-tier applications by program transformation, in: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '05), 2005, pp. 221–232.

[17] E. Cooper, S. Lindley, P. Wadler, J. Yallop, Links: web programming without tiers, in: Proceedings of the 5th international conference on Formal methods for components and objects (FMCO '07), 2007, pp. 266–296.

[18] S. P. Bernard, S. Manuel, An interpreter for server-side hop, in: Proceedings of the 7th symposium on Dynamic languages (DLS '11), 2011, pp. 1–12.

[19] Polyglot Extensible Compiler Framework.
URL `http://www.cs.cornell.edu/projects/polyglot/`

[20] The Javaflow Component.
URL `http://commons.apache.org/sandbox/javaflow/`

[21] G. Wang, T. S. E. Ng, The Impact of Virtualization on Network Performance of Amazon EC2 Data Center, in: Proceedings of the 29th Conference on Information communications (INFOCOM '10), 2010, pp. 1163–1171.

[22] P. Eugster, K. Jayaram, EventJava: An Extension of Java for Event Correlation, in: Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009), 2009, pp. 570–594.

[23] Microsoft Azure.
URL `http://www.microsoft.com/windowsazure/`

[24] L. Barroso, J. Dean, U. Holzle, Web Search for a Planet: The Google Cluster Architecture, Micro, IEEE 23 (2) (2003) pp. 22–28.

[25] M. Neubauer, P. Thiemann, An Implementation of Session Types, in: Proceedings of the 6th ACM International Symposium on Practical Aspects of Declarative Languages (PADL '04), 2004, pp. 56–70.

[26] E. Bonelli, A. Compagnoni, Multipoint Session Types for a Distributed Calculus, in: Proceedings of the 3rd Conference on Trustworthy Global Computing (TGC '07), 2007, pp. 240–256.

[27] A. Bejleri, N. Yoshida, Synchronous Multiparty Session Types, Electronic Notes in Theoretical Computer Science (ENTCS) 241 (2009) pp. 3–33.

[28] S. Gay, V. Vasconcelos, A. Ravara, Session Types for Inter-Process Communication, Tech. rep., University of Glasgow (2003).

[29] A. Vallecillo, V. T. Vasconcelos, A. Ravara, Typing the Behavior of Software Components using Session Types, Fundamenta Informaticae 73 (4) (2006) pp. 583–598.

[30] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, S. Levi, Language Support for Fast and Reliable Message-based Communication in Singularity OS, in: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '06), 2006, pp. 177–190.

[31] M. Dezani-Ciancaglini, N. Yoshida, Asynchronous Session Types and Progress for Object-Oriented Languages, in: Proceedings of the 9th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS '07), 2007, pp. 1–31.

[32] K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, N. Yoshida, Scribbling interactions with a formal foundation., in: Proceedings of the 7th International Conference on Distributed Computing and Internet Technologies (ICDCIT 2011), 2011, pp. 55–75.

[33] A. Tanenbaum, R. Renesse, A Critique of the Remote Procedure Call Paradigm, in: Proceedings of European Teleinformatics Conference (EUTECO '88), 1988, pp. 775–783.

[34] P. Bogle, B. Liskov, Reducing Cross Domain Call Overhead using Batched Futures, in: Proceedings of the 9th ACM Conference on Object-oriented Programming Systems, Language, and Applications (OOPSLA '94), 1994, pp. 341–354.

[35] K. C. Yeung, P. H. J. Kelly, Optimising Java RMI Programs by Communication Restructuring, in: Proceedings of the 4th ACM/IFIP/USENIX 2003 International Conference on Middleware (Middleware '03), 2003, pp. 324–343.

[36] A. W. Appel, Compiling with Continuations, Cambridge University Press, 2007.