Static Analysis of Object References in RMI-based Java Software

Mariana Sharp
Ohio State University

Atanas Rountev
Ohio State University

Abstract

Distributed applications provide numerous advantages related to software performance, reliability, interoperability, and extensibility. This paper focuses on distributed Java programs built with the help of the Remote Method Invocation (RMI) mechanism. We consider points-to analysis for such applications. Points-to analysis determines the objects pointed to by a reference variable or a reference object field. Such information plays a fundamental role as a prerequisite for many other static analyses. We present the first theoretical definition of points-to analysis for RMI-based Java applications, and an algorithm for implementing a flow- and context-insensitive points-to analysis for such applications. We also discuss the use of points-to information for computing call graph information, for understanding data dependencies due to remote memory locations, and for identifying opportunities for improving the performance of object serialization at remote calls. The work described in this paper solves one key problem for static analysis of RMI programs, and provides a starting point for future work on improving the understanding, testing, verification, and performance of RMI-based software.

1 Introduction

Java Remote Method Invocation (RMI) is an object model for developing distributed applications in Java [10]. Using RMI, objects in one Java virtual machine (JVM) can invoke methods on objects in other JVMs. RMI provides powerful features such as object references that cross JVM boundaries, remote invocations that can use entire object graphs as parameters, and distributed garbage collection. RMI can either be used as a stand-alone middleware platform, or as the foundation for more advanced architectures. For example, both Enterprise JavaBeans and Jini are based on RMI and also provide additional middleware services.

Distributed applications play an important role in various commercial, scientific, and engineering domains. The development of such applications poses numerous problems related to software correctness, performance, and maintainability. For RMI applications in particular, some approaches have been investigated for program understanding, performance optimizations, and software testing (e.g., [14, 13, 6, 9, 3, 19, 15]). However, at present there is no work on establishing *systematic foundations for static analysis* of RMI applications. The goal of this paper is to take a significant step towards defining such foundations.

The target of our work is *points-to analysis*. Such analysis determines the objects to which locals, formals, and fields may point. This information has a wide range of uses in other static analyses; in turn, the results of these analyses are used in a variety of program understanding applications, testing approaches, software verification techniques, and performance optimizations. There has been a large body of work on points-to analysis; most of this work is summarized in [7, 17]. However, these existing analyses cannot be applied directly to RMI-based distributed Java applications. Thus, the builders of such applications cannot take advantage of a large number of well-known static analyses (points-to analyses as well as other popular analyses that require points-to information).

Theoretical Model. Our first goal is to establish the foundations for points-to analysis of RMI-based Java applications. We define formally a particular style of points-to analysis: flow- and context-insensitive subset-based analysis (i.e., Andersen-style analysis [1]). Our approach could easily be extended to flow- and context-sensitive points-to analyses, and to analyses that are not subset-based. Such extensions are well understood for non-distributed Java programs (e.g., [5, 12]) and there are no conceptual difficulties in defining such extensions for our analysis.

The importance of these foundations is twofold. First, they provide a basis for defining a wide range of points-to analyses for RMI applications, based on the large number of such analyses for non-distributed programs. Second, they enable work on RMI-based extensions of other popular static analyses (e.g., dependence analyses, side-effect analyses, program slicing, change impact analyses, etc.).

Analysis Algorithm. Our second goal is to define an algorithm for implementing the points-to analysis. The algorithm is a generalization of an approach by Lhoták and Hendren [8] for non-distributed Java programs. We introduce

new techniques that allow the analysis to represent the flow of remote object references, the effects of remote invocations, and the remote propagation of object graphs through serialization. Furthermore, we present an approach for efficient modeling of the code in the standard Java libraries; our experiments indicate that this approach is essential for reducing the running time of the analysis.

Static Analyses for Program Understanding. The third goal of this work is to describe two analyses that use the points-to analysis to enhance the understanding of RMI applications. First, we outline the use of points-to information to identify write-read dependencies due to remote calls. In particular, we consider *inter-component* dependencies, in which components running in two different JVMs potentially access the same memory location. Second, we discuss the use of the points-to analysis to identify opportunities for improving the analyzed program by reducing the cost of serialization at remote calls [19].

Analysis Implementation. Our fourth goal is to implement and evaluate the points-to analysis. We present a preliminary experimental study on a set of 11 RMI applications. Our initial results suggest that the analysis could be a good candidate for a general-purpose points-to analysis of RMI-based programs.

2 Overview of Java RMI

The input to the points-to analysis contains the code for several components $C_1, C_2, \ldots, C_k$. The set of components will be denoted by $\mathcal{C}$. For each component $C_i \in \mathcal{C}$, the analysis takes as input a set $cls(C_i) = \{X_1, \ldots, X_{n_i}\}$ of Java classes. ("Classes" will refer to both Java classes and Java interfaces.) Each component is executed in a separate JVM, typically on a different physical machine. Set $cls(C_i)$ is the complete set of classes that may be loaded at run time in the JVM that executes component C_i. Note that an implementation of the RMI mechanism requires additional helper classes that are generated automatically from classes in $cls(C_i)$. For example, in the default implementation of RMI by Sun, the `rmic` compiler produces a variety of stub classes that implement the details of remote invocations. Such classes are not part of the analysis input.

For any two components C_i and C_j, sets $cls(C_i)$ and $cls(C_j)$ are not necessarily disjoint: it is possible for the same class to be loaded in the two virtual machines that execute C_i and C_j. One example are the classes from the standard Java libraries. We assume that the same version of the libraries is loaded in each JVM; thus, all library classes are included implicitly in $cls(C_i)$ for all $C_i \in \mathcal{C}$.

Figures 1 and 2 show the example used in the rest of the paper; this example is based on a similar example from [4]. For simplicity, we exclude error-handling code (e.g., code related to exceptions thrown by remote invocations). The

```
— Event —
class Event implements Serializable {
     public Date date() { return on; }
     public String description() { return des; }
     public Event(String a) {
       des = a; on = new Date(); }
     private Date on; private String des; }
— Event Listener —
interface Listener extends Remote {
    public void occurred(Event b); }
— Event Channel —
interface Channel extends Remote {
     public void add(Listener c);
     public void announce(Event d); }
```

Figure 1. Running example, part 1.

example contains events, listeners for these events, channels along which events are announced to the listeners, and event sources that create the events and send them to the channels. We consider the following configuration of components:

$$cls(C_1) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyChannel}\}$$
$$cls(C_2) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{MyListener}\}$$
$$cls(C_3) = \{\texttt{Event}, \texttt{Listener}, \texttt{Channel}, \texttt{EventSource}\}$$

In C_1, `MyChannel.main` creates an instance of remote class `MyChannel` and registers it with a naming service. (The naming service will be discussed shortly.) In C_2, `MyListener.main` uses the naming service to obtain a reference to the remote channel object, and then registers with the channel two remote listener objects. Similarly, in C_3, `EventSource.main` obtains a reference to the remote channel object and then announces an event on the channel. In `MyChannel.announce`, the channel object dispatches the event to the registered remote listeners.

2.1 Remote Objects, References, and Calls

A *remote class* implements `java.rmi.Remote`. This is a marker interface that does not contain any methods or fields. A *remote object* is any instance of a remote class. Class `java.rmi.server.UnicastRemoteObject`, which implements `Remote`, provides default support for point-to-point object references using TCP. The simplest mechanism for creating remote classes is to subclass `UnicastRemoteObject`. Other mechanisms are also possible [10], but they are conceptually similar and are beyond the scope of this paper.

A *remote reference* represents a connection between two different JVMs. Similarly to an ordinary (non-remote) object reference, a remote reference is a pointer to an object. The notion of a remote reference is an abstraction: in reality, a component has a reference to a stub object in its own JVM. Typically the existence of these stub objects is ignored, and

```
—— Event Channel Implementation: Component C1 ——
class MyChannel implements Channel
                extends UnicastRemoteObject {
 private Listener[] all; private int num;
 public MyChannel() {
   Listener[] arr = new Listener[10];
   all = arr; num = 0; }
 public void add(Listener c) { all[num++] = c; }
 public void announce(Event d) {
  for(int i=0; i<num; i++) all[i].occurred(d); }
 public static void main(String[] args) {
  String channel_id = args[0];
  Channel e = new MyChannel();
  Naming.bind(channel_id,e); } }

—— Event Listener Implementation: Component C2 ——
class MyListener implements Listener
                 extends UnicastRemoteObject {
 public void occurred(Event b) {...}
 public static void main(String[] args) {
  String channel_id = args[0];
  Channel f = (Channel) Naming.lookup(channel_id);
  Listener g = new MyListener(); f.add(g);
  g = new MyListener(); f.add(g); } }

—— Event Source Implementation: Component C3 ——
class EventSource {
 public static void main(String[] args) {
  String channel_id = args[0];
  Channel h = (Channel) Naming.lookup(channel_id);
  Event k = new Event("abc"); h.announce(k); } }
```

Figure 2. Running example, part 2.

instead RMI programming uses the abstraction of a reference pointing directly to the remote object. An invocation through a remote reference is a *remote invocation*.

Remote references can be created in several ways. For example, a remote invocation can take as an actual parameter an ordinary reference to a locally-created remote object o. As a result of the call, the remotely-invoked method takes as formal parameter a remote reference to o. Another mechanism for obtaining remote references is the use of some *naming service*. The calls to `java.rmi.Naming` in the running example illustrate such use. A naming service is a separate component whose purpose is to allow registration and lookup of remote objects. Sun's RMI implementation provides a default naming service referred to as the *RMI registry*. A call `Naming.bind(name,x)` inserts in the registry a reference to the remote object o referred to by `x`, under the given string name. In the running example, the two invocations `Naming.lookup(channel_id)` are used to initialize local variables `f` and `h` with remote references to the remote object of class `MyChannel`.

While the RMI registry provides a simple naming service, in general there could be other mechanisms for establishing initial "bootstrapping" remote references between two components [10]. Here by "bootstrapping" we mean references that are created with the help of some external mechanism (e.g., a naming service) in order to establish initial connections between components. To model such initial references, we assume that the analysis input contains information about the variables through which such references are created. For each pair of components $(C_i, C_j) \in \mathcal{C} \times \mathcal{C}$, the analysis input contains a set $I_{i \to j}$ of pairs of local variables. Each pair (v_1, v_2) represents a use of the external mechanism which results in creating remote references from v_2 in C_j to all remote objects pointed-to by v_1 in C_i. For our example, $I_{1\to 2} = \{(\mathtt{e}, \mathtt{f})\}$ and $I_{1\to 3} = \{(\mathtt{e}, \mathtt{h})\}$. Sets $I_{i\to j}$ depend on the specific mechanism used by the application. It may be possible to construct these sets automatically in some simpler cases (e.g., when using the default RMI registry). However, since in general the external mechanism for creating initial remote references could be application-specific, programmer input may be required to obtain the information in $I_{i\to j}$.

2.2 Call-by-Copy through Serialization

When actuals of a remote call are references to non-remote objects o_i, the parameter passing mechanism for these actuals is call-by-deep-copy. Objects o_i together with all other objects reachable from them are subject to serialization. This process encodes the object graph starting from o_i and recreates it in the target JVM. For example, consider the call to `announce` in `EventSource.main`. In this call the actual is a (non-remote) reference to an instance o of class `Event`. The class is serializable because it implements the marker interface `java.io.Serializable`. Fields `on` and `des` of o refer to serializable objects. Information about o and the two associated instances of `Date` and `String` is sent across the network. The "mirror image" of this object graph is created in C_1, and formal `d` in `MyChannel.announce` points to the copy of o. This process does *not* invoke the constructor of `Event` in C_1 on the copy of o. The two calls to `occurred` trigger this process again, and in the JVM for C_2 the object graph is recreated twice. Our analysis assumes that objects are serialized using the default serialization mechanism [11], and the application does not use custom serialization methods (e.g., methods such as `writeObject`); this assumption is checked by our implementation.

3 Points-to Analysis

This section defines the theoretical foundations for points-to analysis of RMI-based Java applications. The proposed analysis is subset-based, flow- and context-insensitive, but it should be straightforward to introduce flow sensitivity and various forms of context sensitivity.

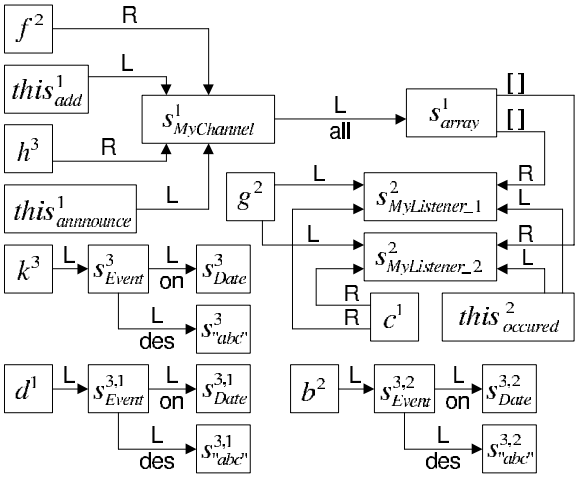

Figure 3. Partial points-to graph.

3.1 Variables, Objects, and Points-to Graphs

The analysis can be defined in terms of several sets. Let Cls be the union of all sets of classes $cls(C_i)$ for all components C_i. We will denote by L the set of all local variables, formal parameters, and implicit parameters `this` in Cls. Similarly, let F and SF be the sets of all instance fields and static fields in Cls, respectively. Finally, let S be the set of all allocation expressions of the form `new X(..)` in Cls.

The analysis is defined in terms of a set V of *variable names* for reference variables, and a set O of *object names* for run-time objects. Figure 3 shows some of these names for the running example. The set V of variable names is a subset of $(L \cup SF) \times \mathcal{C}$. A pair $(v, C_i) \in V$ represents a local variable, a formal parameter, or a static field v in some class from $cls(C_i)$ such that v exists in the JVM executing C_i. The variable names will be denoted by v^i, where the superscript corresponds to the component. For the same $v \in L \cup SF$ there may be multiple $v^i \in V$, each one corresponding to a different C_i.

There are two categories of object names $o \in O$. First, $o = (s, C_i) \in S \times \mathcal{C}$ corresponds to run-time objects that are created by object allocation site s when this site is executed in the JVM for component C_i. Each such object is in the address space of that same JVM. Typically we will use s^i to denote such an object name; as with variable names, the superscript indicates the corresponding component. Each s^i is labeled as remote or non-remote, depending on whether it is an instance of a remote class.

Remote calls can create copies of serializable objects. We use object names $o = (s, C_i, C_j) \in S \times \mathcal{C} \times \mathcal{C}$ to represent such "copy objects". The names will typically be denoted by $s^{i,j}$. Such a name corresponds to a run-time object which exists in the JVM for component C_j and was created as a (transitive) copy of a "normal" object which was created in the JVM for C_i by allocation site s. For example, let s_{Date} be the allocation site `new Date()` in the constructor of `Event` in the running example. Name s^3_{Date} denotes the instance of `Date` which is created in C_3. Due to the remote call to `announce` from C_3 to C_1, a copy of that `Date` object is created in C_1; the name representing this copy object will be $s^{3,1}_{Date}$. The remote calls to `occurred` from C_1 to C_2 create in C_2 two run-time copies of the copy object from C_1. Both objects are transitive copies of the original object from C_3, and are represented by object name $s^{3,2}_{Date}$. Due to the properties of RMI, names $s^{i,j}$ can correspond only to non-remote objects.

The analysis builds a *points-to graph* in which the edges represent points-to relationships. An edge $(v^i, o) \in V \times O$ shows that a variable represented by v^i may point to an object represented by o. An edge $(o_1, f, o_2) \in O \times F \times O$ shows that some object represented by o_1 may store in its f field a reference to an object represented by o_2. An edge (v^i, o) could be either a *remote edge*, denoted by $(v^i, o)_R$, or a *local edge*, denoted by $(v^i, o)_L$. The same subscripts will also be applied to edges (o_1, f, o_2).

For $(v^i, o)_L$ both the variable and the target object must belong to the same JVM. Thus, such edges are either of the form $(v^i, s^i)_L$ or $(v^i, s^{k,i})_L$. Note that s^i could be a remote object (i.e., an instance of a class which implements `Remote`), but the reference to it is still an ordinary local reference. Edge $(v^i, s^j)_R$ represents a points-to relationship through a remote reference, and s^j is always a remote object.[1] Since copy objects created due to serialization cannot be remote, it is not possible to have an edge $(v^i, s^{k,j})_R$. For $(o_1, f, o_2)_L$ the two objects belong to the same JVM; either one (or both) could be a copy object $s^{k,i}$ instead of an ordinary object s^i. For $(o_1, f, o_2)_R$ object o_2 is always a remote object. Figure 3 shows several of the points-to edges for the running example. Edges labeled with `[]` represent points-to relationships for array elements.

3.2 Effects of Program Statements

For brevity, we discuss only the following statements (our implementation handles all other kinds of statements):

- Direct assignment: `v1 = v2`
- Instance field write: `v1.f = v2`
- Instance field read: `v1 = v2.f`
- Object creation: `v = new X`
- Instance invocation: `w = v0.m(v1,...,vk)`

[1] It is possible to have $i = j$ and the reference to be remote at the same time. For example, if C_i creates a remote object, registers it with a naming service, and then immediately looks it up, the component will obtain a remote reference to the object. Calls through this reference will be remote calls that are handled by the RMI infrastructure.

In the above statements, $\mathtt{v}_i \in L \cup SF$ denotes a local variable, a formal parameter (including `this`), or a static field.

The analysis constructs a points-to graph G for the entire application, as well as component-specific sets of reachable methods $Reach_i$ for all $C_i \in \mathcal{C}$. In the beginning, G is empty and each $Reach_i$ contains the main method of the corresponding C_i.[2] For each statement that appears in some method from $Reach_i$ for some i, the analysis adds to G nodes and edges that represent the effects of the statement, and updates all affected sets $Reach_j$.

The rules for handling different statements are represented as function definitions of the form $f(G) = G'$, where G and G' are points-to graphs. The first rule considers the references that are created with the help of an external mechanism such as a naming service:

for each $(v^i, w^j) \in I_{i \to j}$ such that v^i is a local in some $m' \in Reach_i$ and w^j is a local in some $m'' \in Reach_j$:
$f(G) = G \cup \{ (w^j, o)_R \mid (v^i, o)_x \in G \wedge o \text{ is remote} \}$

Points-to edge $(v^i, o)_x$ could be either local or remote: the object exported by component C_i is either created locally, in which case the edge is $(v^i, o)_L$, or is obtained from some other component, in which case the edge is $(v^i, o)_R$.

Suppose the statement under consideration occurs in some method from $Reach_i$ in component C_i.

$$\text{for } v_1 = v_2 : f(G) = G \cup \{ (v_1^i, o)_x \mid (v_2^i, o)_x \in G \}$$

The kind x of the new edge is the same as the kind of the old one ($x \in \{L, R\}$). The sources of the point-to edges are the component-specific copies v_1^i and v_2^i of v_1 and v_2.

$$\begin{aligned}&\text{for } v_1 = v_2.\mathit{fld} : f(G) = G \cup \\ &\qquad \{ (v_1^i, o_2)_x \mid (v_2^i, o_1)_L \in G \wedge (o_1, \mathit{fld}, o_2)_x \in G \} \\ &\text{for } v_1.\mathit{fld} = v_2 : f(G) = G \cup \\ &\qquad \{ (o_1, \mathit{fld}, o_2)_x \mid (v_1^i, o_1)_L \in G \wedge (v_2^i, o_2)_x \in G \}\end{aligned}$$

In reading and writing of object fields, only local points-to edges are considered because fields of remote objects are not accessible through remote references.

$$\text{for } v = new\ X : f(G) = G \cup \{ (v^i, s^i)_L \}$$

Here $s \in S$ is the allocation site corresponding to the `new` expression. Even if the newly-created object is remote (i.e., an instance of a class that implements `Remote`), the reference to it is an ordinary local reference.

$$\begin{aligned}&\text{for } w = v_0.m(v_1, \ldots, v_k) : f(G) = G \cup \\ &\{ \mathit{resolveLocal}(G, m, o, v_1^i, \ldots, v_k^i, w^i) \mid (v_0^i, o)_L \in G \} \cup \\ &\{ \mathit{resolveRemote}(G, m, s^j, v_1^i, \ldots, v_k^i, w^i) \mid (v_0^i, s^j)_R \in G \}\end{aligned}$$

[2] Actually, the initialization of $Reach_i$ should also include all library methods that are executed at JVM startup. Furthermore, during the analysis $Reach_i$ should be updated with static initializers, finalizers, and `run` methods of threads. Our implementation handles these issues.

For calls made through local references we have

$$\begin{aligned}&\mathit{resolveLocal}(G, m, o, v_1^i, \ldots, v_k^i, w^i) \\ &\quad m'(p_0, p_1, \ldots, p_k, ret) = \mathit{dispatch}(o, m) \\ &\quad \text{add } m' \text{ to } Reach_i \\ &\quad \text{return } \{ (p_0^i, o)_L \} \cup \\ &\qquad \{ (p_t^i, o')_x \mid (v_t^i, o')_x \in G \wedge 1 \le t \le k \} \cup \\ &\qquad \{ (w^i, o')_x \mid (ret^i, o')_x \in G \}\end{aligned}$$

The run-time target method m' is determined based on the type of o and on the compile-time target m, using helper function $\mathit{dispatch}$ which encodes the rules for run-time virtual dispatch. The implicit formal `this` in m' is represented by p_0, and the explicit formals are $p_1, \ldots, p_k$. We use ret to denote a special artificial local in m' which is assigned all return values of the method.

For remote invocations from component C_i to a remote object s^j in component C_j, we have:

$$\begin{aligned}&\mathit{resolveRemote}(G, m, s^j, v_1^i, \ldots, v_k^i, w^i) \\ &\ m'(p_0, p_1, \ldots, p_k, ret) = \mathit{dispatch}(s^j, m) \\ &\ \text{add } m' \text{ to } Reach_j \\ &\ \text{return } \{ (p_0^j, s^j)_L \} \cup \\ &\qquad \{ (p_t^j, o')_R \mid (v_t^i, o')_x \in G \wedge 1 \le t \le k \wedge o' \text{ is remote} \} \cup \\ &\qquad \{ (w^i, o')_R \mid (ret^j, o')_x \in G \wedge o' \text{ is remote} \} \cup \\ &\qquad \mathit{resolveSerialization}(G, v_1^i, \ldots, v_k^i, p_1^j, \ldots, p_k^j) \cup \\ &\qquad \mathit{resolveSerialization}(G, ret^j, w^i)\end{aligned}$$

The invoked remote method m' in component C_j is determined based on the same rules for virtual dispatch that are used for ordinary non-remote calls [10]. The invocation creates a local points-to edge from `this` in m' to the remote object s^j. For actual parameters v_t^i that point to remote objects o', remote references to o' are created for the corresponding formals p_t^j of m'. Note that edge (v_t^i, o') could be either local or remote. If the return value of m' is a (local or remote) reference to a remote object o', the left-hand-side variable w^i at the call site starts pointing remotely to o'.

Functions $\mathit{resolveLocal}$ and $\mathit{resolveRemote}$ can be easily augmented to construct the call (multi)graph of the application. The nodes in the call graph are pairs (m, i), where method m belongs to $Reach_i$. The edges correspond to call statements: if statement st in method m in component C_i invokes method n in component C_j ($i = j$ or $i \neq j$), the call graph contains an edge from (m, i) to (n, j), labeled with st. Our implementation builds the call graph on the fly, during the analysis.

3.3 Modeling of Non-Remote Parameters

Function $\mathit{resolveSerialization}$ models parameter passing for non-remote actuals. Recall that for each object name s^i which represents non-remote serializable run-time objects created by allocation site s in component C_i, the analysis defines a set of object names $s^{i,j}$ for copy objects, one

for each component C_j. For convenience, for each component C_j we define the following map μ_j:

- $\mu_j(s^i) = s^{i,j}$ when s^i is a non-remote serializable object created in some C_i
- $\mu_j(s^{k,i}) = s^{k,j}$ when $s^{k,i}$ is an object created in some C_i as a deserialized copy of ordinary object s^k
- $\mu_j(s^i) = s^i$, when s^i is a remote object in some C_i

Given an object name o which represents run-time objects in some component C_i, object name $\mu_j(o)$ represents the corresponding run-time objects in C_j.

The effects of a remote call $v_0.m(v_1, \ldots, v_k)$ on non-remote parameters are as follows. The object graph reachable from $v_1, \ldots, v_k$ is traversed according to the rules described below. All traversed non-remote serializable objects are serialized and recreated in the target component. This process can be described by defining a subgraph $Copied$:

- If $(v_t^i, o)_L \in G$ and o is a non-remote serializable object, then $(v_t^i, o)_L \in Copied$
- If $o \in Copied \wedge (o, fld, o')_L \in G$, where *fld* is a non-transient field and o and o' are non-remote serializable objects, then $(o, fld, o')_L \in Copied$
- If $o \in Copied \wedge (o, fld, o')_x \in G$, where *fld* is a non-transient field, o is a non-remote serializable object, and o' is a remote object, then $(o, fld, o')_x \in Copied$
- $Copied$ is the smallest set with these properties

If a field is declared as transient, its value is not subjected to further serialization. If a non-transient field points to a remote object (either locally or remotely; $x \in \{L, R\}$), the traversal stops and the remote object is not serialized. However, if the field points to a non-remote object, serialization is attempted; if the pointed-to object is not serializable, an exception is thrown. The definition of $Copied$ leads to

$$
\begin{array}{l}
resolveSerialization(G, v_1^i, \ldots, v_k^i, p_1^j, \ldots, p_k^j) = \\
\{(p_t^j, \mu_j(o))_L \mid (v_t^i, o)_L \in Copied \wedge o \text{ is n.r.s.}\} \cup \\
\{(\mu_j(o), fld, \mu_j(o'))_L | (o, fld, o')_L \in Copied \wedge o, o' \text{ are n.r.s.}\} \cup \\
\{(\mu_j(o), fld, \mu_j(o'))_R | (o, fld, o')_x \in Copied \wedge o \text{ is n.r.s. } \wedge \\
\qquad\qquad\qquad\qquad o' \text{ is remote}\}
\end{array}
$$

Here "n.r.s." stands for "non-remote but serializable". The serialization mechanism initializes a copy object (i.e., a deserialized object) not by invoking a constructor of its class, but rather by invoking the no-arguments constructor of the "lowest" non-serializable superclass. It is easy to add this invocation to the rules from above, and for simplicity we omit this detail from the presentation.

4 Analysis Algorithm

This section describes an algorithm for implementing the points-to analysis. Our approach is based on techniques proposed by Lhoták and Hendren [8] for analysis of non-distributed Java applications. For brevity, we discuss the algorithm at a high level, focusing primarily on the new techniques we have introduced in order to handle RMI features.

Pointer Assignment Graph. The analysis uses a data structure referred to as a *pointer assignment graph* (PAG). For a name $v^i \in V$, there is a PAG node $node(v^i)$ corresponding to this name. There are also PAG nodes of the form $node(v^i.fld)$ for each instance field *fld* accessed through v^i. Similarly, for each object name $o \in O$, there are PAG nodes $node(o)$ and $node(o.fld)$.

PAG edges represent flow of values. For example, if a statement $v_1 = v_2$ belongs to some method from $Reach_i$, the PAG contains an edge $node(v_2^i) \longrightarrow node(v_1^i)$. Whenever the analysis adds a method to $Reach_i$, the statements in the body of that method are processed and the corresponding PAG edges are created.

Points-to Sets. PAG edges are used to propagate information about points-to relationships involving the nodes. For each PAG node $node(v^i)$ we define two points-to sets $Pt_L(v^i)$ and $Pt_R(v^i)$ representing the local and remote points-to relationships for v^i. For example, if the PAG contains a non-remote edge $node(v_2^i) \longrightarrow node(v_1^i)$, every element of $Pt_L(v_2^i)$ is propagated to $Pt_L(v_1^i)$, and every element of $Pt_R(v_2^i)$ is propagated to $Pt_R(v_1^i)$.

A remote edge $node(v^i) \stackrel{remote}{\longrightarrow} node(w^j)$ shows that there is a flow of values from a variable v in component C_i to a variable w in component C_j. Such edges are created for the pairs in $I_{i \to j}$ (i.e., for variables used for an external mechanism such as a naming service). Another reason such edges may be created is to represent the flow of values from actuals to formals at remote calls. For a remote PAG edge, the algorithm considers all objects $o \in Pt_L(v^i) \cup Pt_R(v^i)$. If o is a remote object, it is added to $Pt_R(w^j)$. Propagation can also occur when o is a non-remote serializable object, as described below.

Consider a call $w = v_0.m(v_1, \ldots, v_k)$ in some method in $Reach_i$. Suppose that $s^j \in Pt_R(v_0^i)$: that is, v_0^i points to some remote object from component C_j. To model the effects of the remote call for receiver object s^j, the analysis determines the target method m' and adds it to $Reach_j$. Let the formals of m' be $p_0, p_1, \ldots, p_k$ (where p_0 is `this`), and let the return variable of m' be ret. The algorithm

- adds object s^j to $Pt_L(p_0^j)$
- adds edges $node(v_t^i) \stackrel{remote}{\longrightarrow} node(p_t^j)$ to the PAG
- adds edge $node(ret^j) \stackrel{remote}{\longrightarrow} node(w^i)$ to the PAG

The new edges represent the flow of remote references due to parameter passing and return values at remote calls. Subsequently, the analysis may propagate remote objects along these edges. For example, consider the calls `f.add(g)` in `MyListener.main` in the running example. Since the

remote points-to set of f^2 contains $s^1_{MyChannel}$, a remote PAG edge is added from actual g^2 to formal c^1. The propagation of the points-to set of the actual along this edge adds object names $s^2_{MyListener1}$ and $s^2_{MyListener2}$ to $Pt_R(c^1)$, indicating that `c` in C_1 may contain remote references to the two remote instances of `MyListener` created by C_2.

The remote PAG edges created at remote calls also model the effects of object serialization for non-remote actual parameters. To illustrate this process, consider the call to `announce` in `EventSource.main`. The remote PAG edge from actual h^3 to formal d^1 is used to propagate the non-remote serializable s^3_{Event} to C_1. As a result, a copy object $s^{3,1}_{Event}$ is created in C_1 and is added to the local points-to set of d^1. The original object s^3_{Event} has fields `on` and `des` that point to two serializable objects: s^3_{Date} and $s^3_{"abc"}$. The analysis creates remote PAG edges from $s^3_{Event}.des$ to $s^{3,1}_{Event}.des$ and from $s^3_{Event}.on$ to $s^{3,1}_{Event}.on$. Based on these edges, copy objects $s^{3,1}_{Date}$ and $s^{3,1}_{"abc"}$ are created and added to the appropriate points-to sets in C_1. The call to `occurred` from C_1 to C_2 creates additional remote PAG edges; as a result, copy objects $s^{3,2}_{Event}$, $s^{3,2}_{Date}$, and $s^{3,2}_{"abc"}$ are created and propagated to C_2. In the general case, this iterative process is equivalent to function $resolveRemote$ from Section 3.2.

Handling of the Standard Java Libraries. The standard Java libraries are implicitly added to the set of classes $cls(C_i)$ for each component. Based on the analysis definition presented earlier, library variables and objects will have multiple copies. For example, if a library method m has a local variable v, the points-to analysis will use multiple copies of v—that is, a separate name v^i for each component C_i. Object names are treated similarly.

Our initial experiments with this approach showed that the majority of analysis time is spent on processing the relevant code from the libraries. Even when the size of the non-library code is small, the necessary conservative treatment of various features from the libraries (e.g., JVM startup, initialization of static fields, dynamic class loading and reflection, finalizers, etc.) requires the analysis to consider a large number of library methods as reachable. The replication of library variables and objects results in significant running time for the analysis. For example, for a program that was a slightly more elaborate version of the running example, the analysis ran out of memory. For examples containing two components, the analysis running time was around three hours, which was clearly impractical.

To reduce running time, we designed and implemented an alternative technique for handling the standard libraries. The basic idea is to create only one replica of a library entity. For a variable v, the we use a single name v^{lib} instead of multiple names v^i. For an object allocation site s, there is a single object name s^{lib}. The analysis also maintains a set of reachable methods $Reach_{lib}$, and library methods are added to this set rather than to the component sets $Reach_i$.

After the completion of the analysis, the local points-to sets for non-library variables and objects are processed to replace names s^{lib}. For example, if $Pt_L(v^i)$ contains s^{lib}, this object name can stand only for objects created in component C_i; thus, s^{lib} can be replaced by s^i. Note that such a replacement cannot be performed for $Pt_R(v^i)$, because in this case s^{lib} represents objects in any component, and not necessarily objects in C_i.

The full-replication approach from Section 3 and the zero-replication approach from above are the two endpoints of the design spectrum for handling of the standard libraries. Since the degree of replication has a direct effect on both analysis cost and analysis precision, future investigations should be performed in order to understand thoroughly this entire spectrum of cost-precision tradeoffs.

5 Analyses for Program Understanding

Points-to information is a frequently required "enabler" for a wide range of other techniques. This section discusses briefly three specific uses of the points-to analysis for the purposes of program understanding of RMI-based applications. Of course, many other uses are possible (e.g., for program slicing, change impact analysis, etc.).

Call Graph. As discussed Section 3.2, the analysis performs on-the-fly call graph construction. The resulting graph can serve as the starting point for many other static analyses. The call graph can also be used to answer questions such as "Given a call statement st in component C_i, which methods in other components may be invoked by st, directly or transitively?". This and similar questions can enhance the understanding of the inter-method and inter-component flow of control, especially when combined with browsing tools that express the answers visually by displaying graphically the relevant parts of the call graph.

Data Dependencies. Consider a component C_i and some object s^i created in this component. A statement st^i_1 in C_i could potentially read or write some field of s^i (either directly or transitively through its callees). Now consider a call site st^j_2 in some other component C_j, and suppose st^j_2 invokes some remote method from C_i. Due to the remote call, the execution of st^j_2 could (transitively) read or write some field of object s^i. Thus, it is possible to have a read-write or write-read dependence between st^i_1 and st^j_2. The pair (st^i_1, st^j_2) represents a potential *inter-component data dependence* between C_i and the caller component C_j. Furthermore, consider another call site st^k_3 in a third component C_k, and suppose st^k_3 invokes some remote method from C_i. It is possible to have a dependence between st^j_2 and st^k_3 due to some field of s^i. In this case the inter-component dependence is between C_j and C_k, but the memory responsible for the dependence is in the JVM for C_i.

We have defined and implemented an algorithm which, for a given component C_i, computes all pairs (st_1^i, st_2^j) and (st_2^j, st_3^k) that correspond to potential data dependencies, as defined above. For brevity, we will illustrate the algorithm through the running example, rather than presenting a formal definition. Consider component C_1 from Figure 1. The call `h.announce(k)` in `main` in C_3 creates a copy object $s^{3,1}_{Event}$ in C_1 (based on s^3_{Event} in C_3) and initializes its fields *on* and *des* with copy objects $s^{3,1}_{Date}$ and $s^{3,1}_{"abc"}$. Consider now `all[i].occurred(d)` in `announce` in C_1. Since the local points-to set of actual d^1 contains $s^{3,1}_{Event}$, we can determine that due to the serialization, the values of fields $s^{3,1}_{Event}.on$ and $s^{3,1}_{Event}.des$ are read. Thus, there is a write-read dependence between the call to `announce` in C_3 and the call to `occurred` in C_1, due to memory locations $s^{3,1}_{Event}.on$ and $s^{3,1}_{Event}.des$. As another example, consider `f.add(g)` in `main` in C_2 and `h.announce(k)` in `main` in C_3. The calls to `add` results in a modification of $s^1_{MyChannel}.num$, due to `num++`. Since `announce` reads the value of $s^1_{MyChannel}.num$, there is a dependence between `f.add(g)` in C_2 and `h.announce(k)` in C_3.

The computation of such dependencies requires (1) examining local points-to set at reads and writes of expressions $v.fld$, (2) considering the reads and writes of static fields, (3) taking into account the reads and writes performed during object serialization and deserialization, and (4) performing iterative backward propagation of this information on the call graph, from callees to callers.

Customized Serialization. One of the performance bottlenecks for RMI is the serialization and deserialization of non-remote actuals [14, 9]. Several optimizations can be used to reduce this cost. For example, if the types of the serialized objects are unique and known in advance, specialized serialization code can be created rather than using the more expensive default serialization mechanism. As another example, if the object graph that will be serialized is always acyclic, a cheaper version of the serialization algorithm can be used, as opposed to the general version which must detect cycles. Such techniques have been shown to be quite effective in reducing the cost of serialization in RMI applications [19]. By analyzing the structure of the points-to graph produced by our analysis, it is straightforward to expose these optimization opportunities to a programmer. This information enables the introduction of customized serialization, either manually (through methods `writeObject` and `readObject` [11]), or automatically with the help of an optimization tool.

Other Potential Uses. Testing of distributed Java applications can be based on adequacy criteria that consider the coverage of start-to-end scenarios [2]; the corresponding execution paths can be automatically constructed (and monitored at run time) based on the call graph. As another example, the call graph and the data dependencies may be useful for static analyses that attempt to identify potential deadlocks and race conditions in RMI-based Java software.

6 Experimental Study

We implemented the points-to analysis algorithm using the Soot framework [18], version 2.1, and the Spark component of Soot which implements the points-to analysis techniques from [8]. The analysis was executed on a 2.8GHz Pentium4 PC with 1GB of memory. The experiments were performed on the set of RMI-based Java applications listed in Table 1. The applications were obtained from publicly available projects and books, and represent a variety of domains. For example, `auction` implements an auctioning system: clients connect to a server and place bids for items. As another example, `jodl` uses a JOb Dispatching Library to dispatch and execute tasks on different network nodes.

Column (2) shows the number of components C_i in each application, and column (3) contains the sum of the sizes of $cls(C_i)$, excluding library classes. Column (4) describes the number of reachable methods processed by the analysis. Column "User" shows the sum of the sizes of $Reach_i$, and column "All" adds to this number the size of $Reach_{lib}$.

Column (5) shows the running time of the points-to analysis. The number of methods in column "All" is an indication of the amount of work that the analysis needs to perform, since the body of each reachable method must be processed in order to create PAG edges and to "populate" points-to sets. Clearly, the majority of analysis time is spent on processing the relevant code from the standard libraries. As discussed in Section 4, we introduced special handling of library variables and objects in order to reduce analysis cost. As a rough estimate of running time, the cost of the analysis is under 0.05 seconds per analyzed method. The overall analysis time can be reduced further if the libraries are pre-analyzed once and the computed information is reused every time an application is analyzed. Similar approaches have already been developed for points-to analysis for C (e.g., [16]), but it remains to be seen whether they can be successfully adapted to Java.

To gain more insight into the points-to analysis solution, we gathered a variety of measurements. Consider an expression $v.m(\ldots)$ in some non-library method $m' \in Reach_i$. Column (6) shows the total number of such call sites for all components; if a call site occurs in multiple components, it is counted multiple times. Column (7) contains the number of remote call sites—that is, sites for which $Pt_R(v^i)$ was not empty. Most programs have multiple remote call sites, which indicates that there may be several different kinds of remote interactions between application components.

For each site from (7), we computed the number of distinct remote methods that were potentially invoked by the site. More precisely, consider $v.m(\ldots)$ in some method

(1) App	(2) $\#C_i$	(3) Cls	(4) Methods User	(4) Methods All	(5) Time	(6) Calls	(7) Rmt	(8) RmtTrg	(9) RmtRef Param	(9) RmtRef Ret	(10) Serial	(11) OptType	(12) OptCycle
filesrv	2	7	14	7179	5m17s	8	5	1.0	0	0	0	0	0
stocks	2	8	21	7261	5m18s	12	2	1.0	1	0	2	2	2
rmttask	2	9	12	7174	5m14s	9	2	1.0	0	0	1	1	1
channel	3	11	18	7180	5m17s	11	4	1.0	2	0	4	4	4
bank	2	14	21	7186	5m20s	15	9	1.0	0	1	2	2	2
auction	2	17	62	7291	5m26s	75	5	1.0	2	0	4	4	4
jodl	2	29	125	7357	8m38s	158	13	1.0	0	0	2	2	2
jenut	2	33	68	7311	5m35s	99	36	1.0	11	9	20	20	20
translator	2	38	85	7269	5m52s	69	1	2.0	0	0	1	1	1
database	2	67	62	7370	5m59s	33	8	1.0	0	4	2	2	2
ssl	2	67	62	7375	6m01s	33	8	1.0	0	4	2	2	2

Table 1. Experimental Results.

$m' \in Reach_i$. For each $s^j \in Pt_R(v^i)$, let m'' be the target method for receiver s^j. For each remote call site we computed the number of distinct targets m'' based on $Pt_R(v^i)$. Column (8) shows the average number of remote target methods over the call sites from (7). For all applications except one, the analysis resolved each remote call site to a unique target method. Since 1.0 is a lower bound for this metric, these results show that the call graphs contain precise information about the targets of remote calls.

For each remote call site, we also examined the points-to solution and determined whether there is any flow of remote references due to parameter passing. Such flow may occur when there exists an actual parameter v for which $Pt_L(v^i)$ or $Pt_R(v^i)$ contains a remote object. Column (9), subcolumn "Param" shows the number of remote call sites at which remote references may be created in the callee due to actual parameters in the caller. Remote references also may flow as return values in the case when $Pt_L(ret^j)$ or $Pt_R(ret^j)$ contains a remote object; here ret^j denotes the artificial variable that contains the return values of the called remote method. Column (9), subcolumn "Ret" contains the number of remote call sites at which remote references may be created in the caller due to the return value from the callee. The measurements indicate that it is not unusual for RMI applications to create additional remote references at remote calls, either in the callee (through parameter passing) or in the caller (through return values). Thus, any points-to analysis needs to include techniques for handling such flow of remote references. Any subsequent analysis (e.g., change impact analysis) must also take into account this flow, based on the output of the points-to analysis.

We also considered $Pt_L(v^i)$ for an actual parameter v at a remote call site to determine whether serialization for non-remote parameters may occur at the site. Column (10) shows the number of sites from (7) for which serialization may occur due to actuals that point to non-remote serializable objects. These results indicate that RMI applications take advantage of the ability to use serializable objects (and more generally, serializable object graphs) as parameters of remote calls. A points-to analysis cannot expect that the non-remote actual parameters at remote call sites are always of primitive types, and therefore the analysis must model in a general manner the possible effects of serialization. Our analysis handles this issue by introducing remote PAG edges connecting the original object at the caller with its deserialized copy at the callee (Section 4).

The last two columns consider the remote call sites at which serialization may occur (i.e., the sites from column 10). As described in Section 5, points-to information can be used to provide a programmer with information about call sites at which the types of the serialized objects are unique and known in advance, or the object graph that will be serialized is always acyclic. Customized serialization at such call sites can improve the performance of the application. For each site from (10) we determined whether the type-based optimization was possible; the number of optimizable sites is shown in column (11). Similarly, for each site from (10) we determined the shape of the serialized object graph; column (12) shows the number of sites with acyclic graphs. For our subject applications, both optimizations were possible at *all* remote calls at which serialization is performed.

In the future we plan to design the appropriate techniques for pre-analysis of the standard libraries, as well as to obtain additional precision results on more RMI applications. Our long-term goal is to make the analysis a useful, precise, and practical enabler of other static analyses in software tools.

7 Related Work

There is a large body of work on Andersen-style points-to analysis for non-distributed programs, both for C and for Java. The closest related work is the analysis proposed in [8], which serves as the starting point for the PAG-based algorithm in our approach. We introduce various modifications of the techniques from [8]. For example, new kinds of PAG edges and propagation rules associated with them are required for analysis of RMI-based programs. The handling of calls is generalized to simulate the semantics of remote

invocations, including the effects of serialization of object graphs. We also introduce a technique for efficient handling of the standard Java libraries.

There has been very little work on generalizing points-to analyses to RMI-based Java software. The closest related work is [19], where a compile-time points-to analysis is used to optimize the serialization at remote calls in several ways, including the two optimization techniques described earlier. The analysis is described with very little detail, but it appears to be a flow-sensitive and context-insensitive variation of Andersen-style analysis. There is no theoretical definition of the analysis semantics, and no details are given about the algorithms and data structures used to implement this semantics. For example, it is unclear whether the approach uses two different points-to sets (remote and local) per variable, whether component-specific copies v^i of a variable v are used, whether the set of reachable methods is constructed during the analysis or is assumed to be part of the analysis input, and whether the underlying standard libraries are being analyzed. Our work provides a precise theoretical definition as well as specific algorithms and data structures. The experimental results in [19] focus on the effects of the optimizations on performance, while we are primarily interested in uses of the points-to information in tools for software understanding, testing, and verification.

8 Future Work

We consider the work presented in this paper to be a first step in a long-term research agenda for establishing a body of work on static analysis for RMI-based applications. First, obvious targets for future work are various flow- or context-sensitive points-to analyses. Such analyses could be defined as extensions of the approach from Section 3, and their scalability should be investigated carefully. Second, it is necessary to define the RMI-specific generalizations of other categories of analyses such as side-effect analysis, def-use analysis, and escape analysis. Next, these analyses should be evaluated experimentally in the context of program understanding tools (e.g., for change impact analysis) and test coverage tools (e.g., for round-trip-scenario coverage [2]). Finally, it is essential to generalize and evaluate these static analyses for more powerful RMI-based middleware platforms such as Enterprise JavaBeans.

Acknowledgments. We would like to thank the ICSM reviewers for their helpful comments and suggestions.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[2] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[3] S. Ghosh, N. Bawa, S. Goel, and Y. R. Reddy. Validating run-time interactions in distributed Java applications. In *IEEE International Conference on Engineering of Complex Computer Systems*, pages 7–16, 2002.

[4] W. Grosso. *Java RMI*. O'Reilly, 2002.

[5] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems*, 23(6):685–746, Nov. 2001.

[6] B. Haumacher and M. Philippsen. Exploiting object locality in JavaParty, a distributed computing environment for workstation clusters. In *Workshop on Compilers for Parallel Computers*, pages 83–94, June 2001.

[7] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[8] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*, LNCS 2622, pages 153–169, 2003.

[9] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for parallel programming. *ACM Transactions on Programming Languages and Systems*, 23(6):747–775, Nov. 2001.

[10] Sun Microsystems. *RMI Specification*. 2002.

[11] Sun Microsystems. *Serialization Specification*. 2003.

[12] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Transactions on Software Engineering and Methodology*, 14(1):1–41, Jan. 2005.

[13] M. Philippsen and B. Haumacher. Locality optimization in JavaParty by means of static type analysis. *Concurrency: Practice and Experience*, 12(8):613–628, July 2000.

[14] M. Philippsen, B. Haumacher, and C. Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.

[15] B. Quig, J. Rosenberg, and M. Kölling. Supporting interactive invocation of remote services. In *International Conference on Principles and Practice of Programming in Java*, pages 195–200, 2003.

[16] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *International Conference on Compiler Construction*, LNCS 2027, pages 20–36, 2001.

[17] B. G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *International Conference on Compiler Construction*, LNCS 2622, pages 126–137, 2003.

[18] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[19] R. Veldema and M. Philippsen. Compiler optimized remote method invocation. In *IEEE International Conference on Cluster Computing*, pages 127–137, 2003.