Enhancing Remote Method Invocation through Type-Based Static Analysis

Carlo Ghezzi, Vincenzo Martena, and Gian Pietro Picco

Dipartimento di Elettronica e Informazione, Politecnico di Milano
{ghezzi,martena,picco}@elet.polimi.it

Abstract. Distributed applications rely on middleware to enable interaction among remote components. Thus, the overall performance increasingly depends on the interplay between the implementation of application components and the features provided by the middleware. In this paper we analyze Java components interacting through the RMI middleware, and we discuss opportunities for optimizing remote method invocations. Specifically, we discuss how to optimize parameter passing in RMI by adapting fairly standard static program analysis techniques. The paper presents our technique and reports about a proof-of-concept tool enabling the analysis and the subsequent code optimization.

1 Introduction

Two major trends characterize the evolution of software technology during the past decade. On one hand, software applications are increasingly distributed and decentralized. On the other, off-the-shelf components are increasingly used as building blocks in composing distributed applications. The gluing mechanisms that support the assembly of components are provided by the middleware. Although much progress has been achieved in supporting designers while developing distributed applications, it is still true that the level of support provided for traditional centralized software is more mature.

Several techniques are available to optimize code generation for predefined target architectures. Today's challenge, however, is to deal with distributed applications where a conventional programming language is used to develop components and a middleware layer is used to interconnect them. We are not aware of optimization techniques that span over the two domains—i.e., the programming language and the middleware. This is true also when the two are in the same linguistic framework, as in the case of Java and RMI [10].

Providing these optimizations is precisely the goal of our work. We concentrate on parameter passing across network boundaries, when object methods are invoked remotely and parameters are serialized. Serialization may introduce a serious performance overhead for large objects. Furthermore, often only a small part of an object is actually used remotely. In these cases, performance would greatly improve if only the used portion of the object were transmitted.

In this paper we discuss how to achieve this optimization by statically analyzing the bytecode of a Java program, and then using the results to optimize the

run-time object serialization. We use fairly standard static analysis techniques. What is new in this paper is the way static analysis is used. Our technique is particularly valuable when off-the-shelf components are used to build a distributed application. In this case, the designer has no visibility of the internals of the components, and therefore many opportunities for hand-optimizing inter-component interactions are necessarily missed. Moreover, since our technique is aimed at reducing the communication overhead, it is particularly useful in bandwidth-constrained scenarios, such as mobile computing.

The paper is structured as follows. Section 2 provides an overview of RMI. Section 3 defines the problem and introduces a reference example. Section 4 describes our program analysis approach. Section 5 reports on a proof-of-concept tool we developed to support our approach. Section 6 discusses limitations and extensions for our technique. Section 7 briefly surveys related work. Section 8 ends the paper with brief concluding remarks.

2 Background

In this section we present the basics of serialization and RMI that are relevant to our work.

2.1 Object Serialization

Serialization is the process of flattening a structured object into a stream of bytes. It provides the basic mechanism to support I/O operations on objects, e.g. to save them on persistent storage, or to transfer them across the network.

Serialization is accomplished in Java by using two I/O streams, `ObjectInputStream` and `ObjectOutputStream`. When an object reference r is written to the latter, the run-time recursively serializes its attributes until the whole graph of objects rooted at r is serialized. In this process, primitive type attributes (e.g. `int`) and `null` attributes are serialized by using a default format. Class descriptors are also inserted in the serialization stream to provide the receiving side with enough information to locate the correct type at deserialization time. Deserialization essentially proceeds backwards, by extracting information from the serialization stream and reconstructing the object graph accordingly. Interestingly, serialization preserves aliases within a single serialization stream.

The aforementioned process requires r and all the other object references to belong to a type implementing the `java.io.Serializable` interface. The programmer, however, retains control over the fraction of the object graph that must be serialized. Although by default all the object attributes are serialized, attributes that are prepended by the `transient` modifier are not. When the object is reconstructed by deserialization, transient attributes are set to the language default for their type.

2.2 Remote Method Invocation

In RMI, a line is drawn between *remote objects* and *non-remote objects*. A "remote object is one whose methods can be invoked from another Java virtual

machine, potentially on a different host" ([10], §2.2). Remote objects are defined programmatically by any class that implements the `java.rmi.Remote` interface or a subtype thereof. All the other objects are simply called non-remote objects.

A reference to a remote object can be acquired by querying a lookup service—or *registry* in the RMI jargon. The registry is a process that binds local objects to symbolic names. Remote clients can query the registry by providing a symbolic name, and obtain a network reference to the corresponding object.

A reference to a remote object can also be obtained through parameter passing. Object parameters can be passed in a remote method invocation either by reference or by copy. If the object being passed is a remote object and it has been exported to the RMI run-time, then the object is passed by reference, i.e., it is accessed through the network. Instead, if the parameter is a non-remote object, or a non-exported remote object, it is passed by copy. In this case, however, the type of the object is required to implement the interface `java.io.Serializable`. Primitive types are always passed by copy.

The semantics of parameter passing *by copy* is defined in Java RMI by object serialization. The interplay between serialization and parameter passing, however, slightly complicates the picture. The first issue is aliasing. Since a single serialization stream per remote method invocation is used, references to the same object in the caller are mapped in the callee into references to the same serialized copy of that object. Hence, parameters are not copied independently, as usually happens in parameter passing by copy. The other issue has to do with serialized objects containing references to remote objects. In this case, the behavior of RMI is as follows ([10], §2.6.5):

- If the object being serialized is an instance of `Remote` and the object is exported to the RMI run-time, the stub for the remote object is automatically inserted in the serialization stream in place of the original object.
- If the object is an instance of `Remote` and the object is not exported to the RMI run-time, or the object is not an instance of `Remote`, the object is simply inserted in the serialization stream, provided that it implements `Serializable`.

In essence, this preserves the semantics of object references in presence of distribution. If the object o contains a remote object r in its object graph, the serialized copy of o still accesses the original copy of r on the original node, if r has been exported. Otherwise, r is treated just like any other ordinary object.

3 Motivation and Reference Example

Object serialization can be the source of severe inefficiencies in remote method invocations. In Java—and object-oriented languages in general—objects are often highly structured: composition quickly leads to pretty large object graphs. If the object must be transferred to another host by a remote method invocation, performance may be affected severely. An overhead is introduced in terms of both *computation*, as (de)serialization requires a recursive navigation of the

object graph, and *communication*, as large objects result in large serialization streams being transmitted. Most of the existing approaches focus on reducing only the computational overhead [1, 2, 7, 8, 12]. They aim at improving the middleware run-time without considering the application code exploiting it and, in particular, how the object is used after deserialization. In this paper, we take the complementary approach of optimizing serialization to reduce communication overhead, based on how serialized objects are used on the server side.

Our goal is to optimize parameter passing for each remote invocation, by serializing only a portion of the object and hence reducing the network traffic. In fact, different invocations may access different portions of the remote objects passed as serialized parameters. The proposed solution consists of (a) performing static analysis to derive information about the portions of serializable parameters that must be transmitted at each call point, and (b) using this information at run-time to optimize serialization.

Let us consider a simple reference example[1], used throughout the paper. A simple print service is provided by the `IPrinter` interface shown in Fig. 1. Clients are expected to invoke the method `print()` by passing the page to be printed as a parameter. Pages are made up of page elements; their interfaces are also shown in Fig. 1. Pages can contain text and/or graphical elements. The methods exported by the `IPage` interface allow one to retrieve either or both. `IPageItem` exports a method `print()`, which is invoked by the receiving `IPrinter` and causes the actual printing of the element on the device.

```
public interface IPrinter
  extends Remote {
  public void print(IPage page)
    throws RemoteException;
}
public interface IPage
  extends Serializable {
  public IPageItem[] getWholePage();
  public IPageItem[] getTextItems();
  public IPageItem[] getGraphicItems();
}
public interface IPageItem
  extends Serializable {
  public void print(PrintStream ps);
}
```

Fig. 1. Interfaces of a simple print service.

In our example, several implementations of `IPrinter` and `IPage` exist, corresponding to different kinds of printing devices and of pages. Sample implementations (`DotMatrixPrinter` and `MixedPage`) are shown in Fig. 2. The client of the printing service, however, ignores the kind of remote printer that is actually being used; it simply invokes the print service. It is up to the server to perform the requested service according to its own capabilities. For example, a dot-matrix printer only prints the textual part of the page (see Fig. 2). Let us consider the case where a composite page is to be printed on a printer that happens to be a dot-matrix printer. Although only textual elements are going to be accessed by the printer, all page elements are serialized and transferred to the server. This is an example where serialization and transmission of a large unused portion of an object generates unnecessary computational and communication overhead, because the server only refers to a portion of the data. The client has no means to avoid this unnecessary serialization—unless information hiding is broken.

We can draw generalized remarks from this example. Often the client has no control over the server's behavior. The server may change over time, due to

[1] The complete source code of the example can be found in [4].

```
public class DotMatrixPrinter
  extends UnicastRemoteObject
  implements IPrinter {
  private PrintStream out = new DotMatrixPS();
  public DotMatrixPrinter()
    throws RemoteException { super(); }
  public void print(IPage page)
    throws RemoteException {
  IPageItem[] text = page.getTextItems();
  if (text != null)
    for (int i = 0; i < text.length; i++)
      text[i].print(out);
  }
}
```

```
public class MixedPage
  implements IPage{
  private IPageItem[]pageItems;
  public IPageItem[]getTextItems(){
    TextItem[] text=
      new TextItem[pageItems.length];
    int j=0;
    for(int i=0;
        i<pageItems.length;
        i++)
    if (pageItems[i]
          instanceof TextItem)
          text[j++]=(TextItem)pageItems[i];
    return text;
  }
  ...
}
```

Fig. 2. Sample implementations of a printer (left) and a page (right).

dynamic binding, and different servers, though presenting the same interface, may differ in their internal behaviors. Internal behaviors are not visible, either because of a deliberate design choice, as in our example, or because they are hard-wired in an off-the-shelf component, whose implementation is responsibility of a third party and hence outside the designer's control.

The next section presents a program analysis technique that enables run-time optimization of remote method invocations. Situations like the one we described can be detected automatically during a static analysis phase, to determine which fields of a given object involved in a remote method invocation are actually used by the target and which are not. The results of analysis can be exploited by automatically generating code that selects the fields to be serialized for each invocation, skipping the serialization of fields unused on the server side. Needless to say, our technique does preserves correctness of the application, i.e., it guarantees that there will never be an attempt to access an object field that has not been serialized.

4 Type-Based Static Analysis

This section describes our static analysis technique in a stepwise manner. We begin by describing the overall analysis strategy, then introduce the notion of concrete graph, which is central to our approach, and conclude by describing the details of the analysis.

4.1 Overall View of the Method

For each remote method invocation `r=o.m(`$p_1, \ldots, p_n$`)` and for each serializable parameter $p_i \in \{r, p_1, \ldots, p_n\}$, we identify which attributes of p_i need to be copied through serialization and passed to the remote target object. To achieve this goal, we focus our analysis on the types that can be instantiated through a `new` operation. These types, called *concrete types*, include all primitive types, and do not include any abstract class or interface.

Our analysis technique is structured in the following phases:

1. Given the overall set of types constituting the application, determine:
   (a) the set $\mathcal{R}$ of *remote types*, i.e., types that extend or implement `Remote`;
   (b) the set $\mathcal{S}$ of *serializable types*. This includes primitive types (e.g., `int`) and reference types that extend or implement `Serializable`. The declaration of an array `T[]` causes the insertion in $\mathcal{S}$ of both `T` and the type "array of `T`".
2. Compute the set of *concrete* remote and serializable types, i.e., the subsets $\mathcal{R}_c \subseteq \mathcal{R}$ and $\mathcal{S}_c \subseteq \mathcal{S}$ containing only concrete types.
3. For each class $c \in \mathcal{R}_c$, identify the set $\mathcal{M}$ of methods that can be invoked remotely. This includes all the methods belonging to the interfaces which extend `Remote` and implemented by c.
4. For each remote invocation of a method $m \in \mathcal{M}$, identify the set of parameters $\mathcal{P}_m$ that must be serialized, i.e., those for which at least one dynamic type belongs to $\mathcal{S}_c$.
5. For each parameter $p \in \mathcal{P}_m$, identify the attributes of p for which serialization can be safely skipped.

Phases 1-4 are quite straightforward, and can be accomplished by inspecting the code and the inheritance hierarchy. Fig. 3 shows their result for our reference example. Phase 5 is the most complex and constitutes the core of our analysis. In a remote method invocation `r=o.m`$(p_1, \ldots, p_n)$, the analysis is complicated by polymorphism. In fact, a parameter p_i of static type T can be replaced at run-time by any subtype of T. The same holds, recursively, for every attribute of T. For each parameter p_i and for each attribute a potentially reachable from it, we must determine whether a is used, and hence it must be serialized, or instead we can safely avoid to do so. The next two sections describe how this can be accomplished.

$\mathcal{R}$ = {IPrinter, DotMatrixPrinter}
$\mathcal{R}_c$ = {DotMatrixPrinter}
$\mathcal{S}$ = {IPage,MixedPage, IPageItem,IPageItem[], TextItem,TextItem[], GraphicItem,GraphicItem[], int,int[],int[][],char,char[]}
$\mathcal{S}_c$ = { MixedPage, TextItem,TextItem[], GraphicItem,GraphicItem[], int,int[],int[][],char,char[]}
$\mathcal{M}$ = {print}
$\mathcal{P}_{print}$ = {aPage}

Fig. 3. The relevant sets for the reference example.

4.2 Concrete Graphs

Each parameter of a remote method invocation can be associated with one or more descriptors, called concrete graphs. Intuitively, a concrete graph associated with a reference parameter p of type T is a directed multi-graph that represents the type structure of one of the possible instances of p at runtime, according to the class hierarchy. The nodes of the concrete graph are serializable types belonging to $\mathcal{S}_c$. Each edge departs from a node representing the type of an object, ends in the node representing the type of one of the object's attributes, and is labeled with the name[2] of such attribute.

[2] Attribute names must be fully specified, i.e., include the type where they are defined. Hereafter, we use only the attribute label as it is unambiguous in our example.

The concrete graphs for a given parameter are computed through an inspection of the static class hierarchy. Fig. 4 shows the two concrete graphs for a parameter of `MixedPage` type shown in Fig. 2. The two concrete graphs differ in the concrete type of the array attribute `pageItems` of `MixedPage`. The graph in Fig. 4(a) describes the case where an element of the array[3] is of type `TextItem` (a character array). The other graph, in Fig. 4(b), describes the case where the element is instead of type `GraphicItem` (an integer matrix)[4].

Formally, a concrete graph is a tuple $\langle \mathcal{N}, \mathcal{E}, \mathcal{A}, \mathcal{S}_c, \mathit{type}, \mathit{attr} \rangle$. $\mathcal{N}$ and $\mathcal{E}$ are, respectively, the set of nodes and edges of the graph, with $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times \mathcal{A}$, being $\mathcal{A}$ the set of attribute names. $\mathcal{S}_c$ is the set of serializable concrete types. Functions *type* and *attr* represent the object structure:

$$\mathit{type} : \mathcal{N} \rightarrow \mathcal{S}_c \qquad \mathit{attr} : \mathcal{E} \rightarrow \mathcal{A}$$

Function *type* is such that $\nexists n_1, n_2 \mid \mathit{type}(n_1) = \mathit{type}(n_2)$ i.e., each serializable type appears in the concrete graph exactly once. Function *attr* yields the name associated with an edge. For instance, if n_1 and n_2 are the first two nodes of the concrete graph in Fig. 4(a), and e the first edge, then $\mathit{type}(n_1) =$ `MixedPage` and $\mathit{attr}(e) =$ `pageItems`.

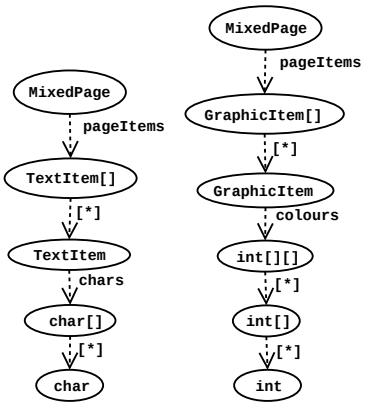


Fig. 4. The concrete graphs of a `MixedPage` parameter.

Intuitively, concrete graphs are used as follows. First, we assume that, for all remote invocations, each serializable parameter has its associated set of concrete graphs. Static analysis is then performed by examining each concrete graph and determining, for each field, if it is used on the receiving side and hence should be serialized[5]. This information is recorded by properly annotating the edges of the concrete graph, and can be exploited at run-time, when the actual dynamic type of each node is known, to determine whether to serialize or skip a given attribute.

4.3 The Analysis in Detail

In this section we provide a detailed description of the core of our technique, i.e., phase 5 of the program analysis described in Section 4.1. To simplify the presentation, we focus on method invocations with a single input parameter and no return parameters. Method signatures with arbitrary arity and types, and encompassing serializable return parameters, can be treated straightforwardly[6].

[3] An edge labelled `[*]` denotes indexing in the array.

[4] Clearly, the array attribute `pageItems` can in general contain any combination of the two.

[5] Clearly, in the case of recursive types only an approximation is possible.

[6] A simple way to deal with multiple parameters is to represent them as attributes of a fake, single parameter. As for the return value, it is sufficient to analyze the

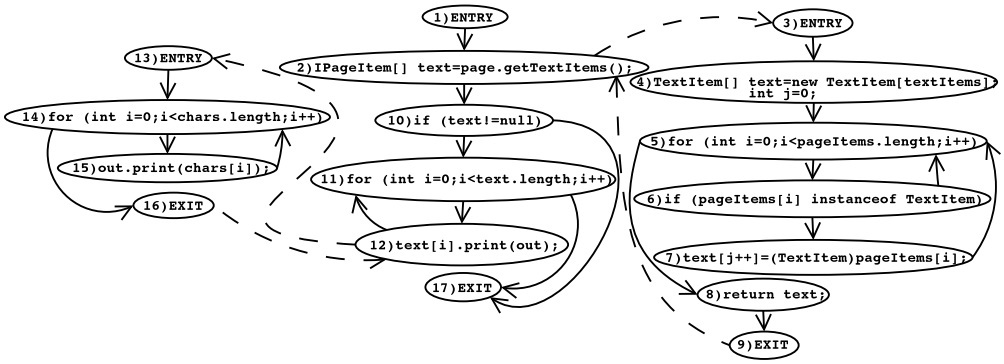


Fig. 5. The control flow graph of the methods `DotMatrixPrinter.print`, `MixedPage.getTextItems`, and `TextItem.print`.

Exploiting the concrete graph The analysis of a method `m(p)` starts by building the concrete graphs associated to p. Then, it analyzes the control flow of `m`. As the analysis walks through the body of `m`, it "decorates" each concrete graph of p by keeping track of whether a given attribute can be serialized or not, based on how the control flow has used the attribute thus far. This information is derived incrementally as the control flow is examined, and relies on the definition of two labelling functions mapping each edge of a concrete graph to a boolean value:

$$\mathit{defined} : \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\} \qquad \mathit{skip} : \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\}$$

The value returned by $\mathit{defined}(e)$ is *true* if the attribute associated with the edge e (i.e., $\mathit{attr}(e)$) has been already assigned a value at a given point in the analysis. The value of $\mathit{skip}(e)$ is *true* if the attribute $\mathit{attr}(e)$ can be safely skipped during the serialization process of the parameter p associated to the concrete graph.

Analyzing the control flow To inspect the control flow of the invoked method, we follow the standard data-flow framework described in [11], which relies on a *control flow graph,* where nodes represent program statements and edges represent the transfer of control from one statement to another. As an example, Fig. 5 shows the control flow graph for the methods `print` in `DotMatrixPrinter`, `getTextItems` in `MixedPage` and `print` in `TextItem` (both invoked by `print` in `DotMatrixPrinter`). The control flow graph of each method starts with an entry node and ends with an exit node. Hence, the overall program control flow can be built out of the method control flow graphs by moving from one control flow graph to the other according to method invocation and termination[7].

Program analysis is carried out by relying on two groups of equations. The first group focuses on a given node in the control flow graph, and defines the

client code (instead of the server) using the return value as the parameter driving the analysis.

[7] Exception-handling introduces additional implicit control transfers. However, these can be analyzed by using existing techniques (e.g., [13]) in conjunction with ours.

relation between the information entering and exiting the node. This group of equations is sufficient to analyze a single path in the given program. However, a node in the control flow graph may have multiple incoming edges that represent different control flow paths, e.g., due to branches or loops, as shown in Fig. 5. The second group of equations specifies precisely how the information coming from these different sources is merged at the entry point of a given target node n, by defining the relationship between the outgoing information associated to the sources of all the edges insisting on n, and the information effectively entering n. Given these two groups of data-flow equations, the global solution can be computed with a standard *worklist* algorithm (see, e.g., Chapter 6 of [11]).

Object aliasing Our analysis is complicated further by *object aliasing*, i.e., the ability of Java to refer to the same object through different references. To determine if an object must be serialized, we must keep track of all the uses of its aliases. The aliasing problem is widely studied and can be tackled by a number of techniques (e.g., [9,14]). Moreover, alias analysis is orthogonal to the type-based analysis described here, and the two can be combined as follows. Given a concrete graph, we first exploit the results of alias analysis to annotate each node of the control flow graph with the alias set associated with each attribute found in the concrete graph. Then, when we "decorate" the edges of the concrete graph, we change the state of an edge not only when an attribute is being modified by a node of the control flow graph, but also when any of its aliases is.

In the sequel, we first describe how the analysis is performed on a single path, by defining how each instruction in the control flow graph of `m` affects the labeling functions *defined* and *skip*. Then, we explain how these functions are "merged" when paths on the control flow graph meet.

Analyzing a Single Path To simplify the presentation, we assume that the input parameter p in the method invocations `o.m(p)` has a single concrete graph and is serializable, i.e., $p \in \mathcal{S}_c$. Moreover, we assume that all multiple-level reference expression such as `a.b().c()` and `a.b.c` are normalized into a sequence of two-level reference expression of the form `a.b()` and `a.b`, by using additional variables. For instance, `y=a.b().c()` can be split in `x=a.b(); y=x.c()`. In the initial state, $\mathit{skip}(e) = \mathit{true}$ and $\mathit{defined}(e) = \mathit{false}$, $\forall e \in \mathcal{E}$, where $\mathcal{E}$ is set of edges belonging to the concrete graph of p. That is, all the attributes of p are undefined and their serialization can be skipped.

We focus the discussion on a variable y being analyzed in the context of the execution of the given method m, where y is either represented in the concrete graph by some edge e such that $y = \mathit{attr}(e)$ with $e = (n_i, n_j, v)$ and $v = y$, or it is an alias of the variable v represented by e.

Let us specify how the traversal of a given node of the control flow graph involving y affects the concrete graph, and in particular the labelling of its edges. Variable y can be affected by definitions and uses (in the common meaning of program analysis [6, 15]). *Definitions* of y are statements which assign a new

value to y. *Uses* of y are all those situations where y's value (or one of y's attribute values) is used in an expression.

The data-flow equations can then be expressed informally as follows:

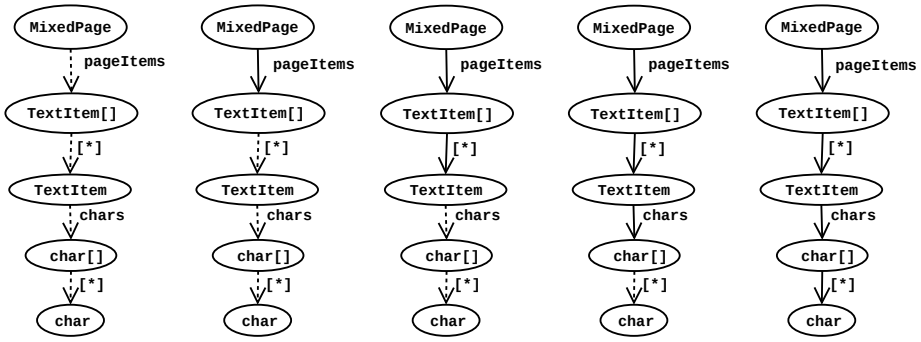
- *Definition.* If *defined*(e) is *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done, since y was already defined and the state of the concrete graph up to date. Otherwise, the value of *defined* is set to *true* for e.
- *Use.* If *defined*(e) is *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done. Otherwise, the value of *skip*(e) must be set to *false*, since the value of y is needed in the execution of the method under analysis.

Attribute accesses and method invocations are an important kind of use. An *attribute access* to y is in the form[8] `y.x`, where x is an attribute defined in the class of y. It requires to consider, from this point in the analysis on, not only the definitions and uses of y but also those of x, to determine whether it is in turn serializable. *Method invocations* involving y can be either[9] of the form `y.g(...)` or `g(y,...)`. Method invocations can be treated as uses, but they require also method g to be analyzed, by operating on the same concrete graph that was labelled up to the invocation point.

Example Let us consider the example of Section 3 and the remote invocation of `print` on an object of type `DotMatrixPrinter`, with a parameter of type `MixedPage`. The concrete graphs for this case are shown in Fig. 4, and the control flow graphs are shown in Fig. 5. Let us consider the concrete graph of Fig. 4(a) and walk through all paths of the control flow graph. Fig. 6 shows the result of this analysis as a series of snapshots of the concrete graph as the control flow graph is analyzed. A dashed edge e means that the corresponding attribute can be safely skipped, i.e., *skip*(e) = *true*, while a solid edge means that the attribute must be serialized.

The analysis of the control flow graph begins in node 1 of Fig. 5. Upon entering the first statement of `print` in node 2, the formal parameter `page` (and hence the actual parameter) is used during the invocation of method `getTextItems`. Invocation of this method is analyzed by moving to the entry point of its control flow graph (node 3), but keeping the same concrete graph. The traversal of node 4 leaves the concrete graph unaffected. On the other hand, node 5 contains a use of the attribute `pageItems`, through access to its attribute `length`. The edge corresponding to `pageItems` is then marked as to be serialized, shown with a solid arrow in Fig. 6(b). Node 6 must be considered next. This node is an interesting case since it illustrates how the construct `instanceof` must be handled. Although `pageItems[i]` is an argument of this instruction, this is not a use of the variable. The result of `instanceof` does not depend on the *value* of `pageItems[i]`, but only on its *type*. Moreover, the value of this variable is left

[8] As mentioned earlier, if y is an array then $y[i]$ is treated as a reference to an attribute.
[9] The `new` instructions can be viewed as a special case of method invocation.



(a) node 1 (b) node 5 (c) node 12 (d) node 14 (e) node 15

Fig. 6. Decorating the concrete graph of Fig. 4(a) while walking through the control flow graphs in Fig. 5. The value of *skip* is *true* for dashed edges and *false* for solid ones.

unaffected by the execution of `instanceof`. Hence, the traversal of this node leaves the concrete graph unchanged. Note also that in this case we are *forced* to go through node 7 instead of choosing the `else` branch and return to node 5. In fact, choosing the latter path would yield to a violation of the previous assumption about the type of the elements of `pageItems`.

The remaining two nodes of `getTextItems` do not affect the concrete graph directly, but establish the object aliases that enable further changes effected by the other methods. Node 7 establishes an alias between an element of `pageItems` and an element of the local array `text`. Node 8 propagates this alias back to the caller `print`, by returning `text` as a result value. Node 9 brings the control back to `print`, which resumes from node 10. Nodes 10 and 11 do not affect the concrete graph, since they contain only uses of `text`. Node 12 contains a use of an element of `text`, which is potentially aliased to one of `pageItems`. Hence, the corresponding edge in the concrete graph must be marked accordingly, as in Fig. 6(c). Such use is a method invocation, which causes the analysis to move to the control flow graph of `print` (node 13).

The first statement of this method (node 14) contains a use of the array `chars` which, by virtue of aliasing, is an attribute of the element of `pageItem` aliased to the invocation target `text[i]`. Hence, `chars` must be serialized (Fig. 6(d)). Finally, node 15 contains an invocation of the method responsible for printing an element `chars[i]`. Although here we do not show the code of this method, intuitively it relies on the input parameter, which then needs to be serialized, leading to the last and final concrete graph in Fig. 6(e).

According to this analysis, the whole object graph of the parameter must be serialized. In our example this matches intuition, since all the information associated to a text page is actually used by a dot-matrix printer.

Let us examine now what happens if the concrete graph of Fig. 4(b) is considered instead, when walking through the same control flow graphs in Fig. 5. Up to node 6, the analysis proceeds as in the previous case, by requiring the attribute `pageItems` to be serialized. The test in node 6, however, forces us to choose a different path, and return to node 5. In fact, proceeding to node 7 would violate the assumption that the elements of `pageItems` are of type `GraphicItem`. The rest of the analysis proceeds through nodes 5, and 8 to 17. However, since no alias has been established between `text` and some attribute of the concrete graph, the latter remains unchanged: only the edge between nodes `MixedPage` and `GraphicItem[]` in Fig. 4(b) become solid. Hence, the analysis confirms our intuition that serialization of an element of `pageItems` whose type is `GraphicItem` can be safely skipped.

Merging Information from Multiple Paths What we described thus far is sufficient to analyze methods whose code does not contain branches in the control flow. Otherwise, we need to specify how the information collected through separate control paths is merged when the control paths are rejoined. Such information is the labelling of edges of the concrete graph, i.e., the value returned by the functions *defined* and *skip*. The problem is that an attribute y in the concrete graph may have been recorded as defined ($\mathit{defined}(e) = \mathit{true}, y = \mathit{attr}(e)$) through one control path, and not in another. Even worse, the same attribute may have been deemed necessary to the enclosing method, and hence marked as to be serialized along one path, and marked as to be skipped along another.

Clearly, to preserve a correct program behavior we need to take the most conservative stand. In the aforementioned case we need to preserve, in the node where the control flow rejoins, the values $\mathit{defined}(e) = \mathit{false}$ and $\mathit{skip}(e) = \mathit{false}$. In other words, an attribute is defined in the joining node if it was defined through all of the joining paths, and similarly it can be safely skipped during serialization if it can be skipped through all the joining paths. Formally, if $\mathit{defined}_i$ and skip_i are those computed along an incoming path i, $\forall e \in \mathcal{E}$:

$$\mathit{defined}(e) = \bigwedge_{i=1}^{n} \mathit{defined}_i(e) \qquad \mathit{skip}(e) = \bigwedge_{i=1}^{n} \mathit{skip}_i(e)$$

Once data-flow equations are given, the analysis is completely defined and the least solution can be computed by the worklist algorithm. The analysis must be performed for each method that can be invoked remotely, for each serializable object parameter, and for each of the possible concrete graphs of such parameter.

5 Prototype

We developed a toolkit to support the approach described in this paper. The overall architecture is showed in Fig. 7. The core component is the *analyzer*, which receives a Java source code as input and outputs information about each remote method invocation, with the corresponding annotated concrete graphs.

The output is in binary format for the sake of compactness. The current implementation of the analyzer is built on top of JABA [5], an API supporting program analysis of Java bytecode.

The result of the analysis can be input to one of three tools: The serialization *checker*, which detects all unused fields declared as serialized, as mentioned in Section 3. The *viewer*, which enables visualization of the analysis output in a human readable format. The *optimizer*, which exploits the analysis results to instrument the source code, by properly redefining serialization. The instrumentation process is non-invasive: it preserves user-defined serialization (when present), and does not affect other uses of serialization (e.g., for storing an object in a file). Details are available in the full technical report [4].

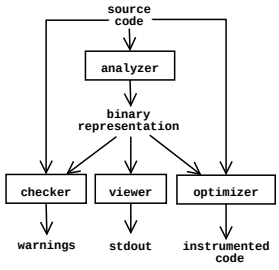


Fig. 7. Tool architecture.

6 Discussion

In this section we discuss improvements and limitations of our technique.

Semi-static analysis Program analysis is usually performed statically, and indeed this is the way our approach works, too. The reason is that the computational load is often too high to be placed on the run-time system. Nevertheless, our main goal is to reduce bandwidth utilization. Hence, in some cases (e.g., in mobile environments with low-bandwidth links) it is reasonable to trade computation for bandwidth, and perform some if not all of our analysis at run-time.

The advantage of this approach lies in the accuracy of information about the program that becomes available at run-time. For instance, if the analysis were to be performed right upon a remote method invocation there would be no need to consider *all* the possible combinations of concrete graphs for a given parameter and control flow graphs of the possible servers. Considering the single concrete graph matching the parameter being passed and the specific server target of the invocation would be sufficient. Hence, while on one hand there is a computational overhead to be paid at run-time, this overhead would arguably be significantly smaller than the one to be paid by an entirely static analysis.

Closed vs. open world We implicitly assumed that the whole code base of the distributed application is available for analysis, and it is not going to change after the code is deployed. This *"closed world"* scenario holds for a number of distributed applications. RMI, however, was designed to support an *"open world"* scenario where the application code base can change dynamically and seamlessly, by virtue of encapsulation and mobile code. In this case, our analysis would no longer be applicable as is. Let us assume that, in our example, `IPrinter` exports an additional method `getPage` returning the page currently being printed. This method can be invoked by a client C_2, different from the client C_1 that required the page printout. No assumption can be made in general about how C_2 uses the page. For instance C_2 might require the serialization of the entire page as

originally stated by the programmer. Now the question is whether the code of C_2 is available at the time of the analysis. If so, the need to serialize all the fields of a page is discovered when the analysis is run on the remote method invocation of `getPage` issued by C_2. Instead, if the code of the clients that may invoke `getPage` is not available, our approach does not work.

We are currently extending our analysis to encompass an open world scenario. This can be achieved by exploiting escape analysis [3], to determine if a given object cannot *escape* the code base known at analysis time. In case it is, the result of our analysis is still valid as is, since an object that has been only partly serialized is never passed outside the boundaries of the analyzed code. Instead, if an object escapes such boundaries no assumption can be made about its use.

7 Related Work

The existing approaches to RMI optimization focus on optimizing the computational overhead of serialization, rather than its bandwidth consumption. Most of these approaches are intended for scientific applications exploiting parallel computing, where computational efficiency is the main concern. To the best of our knowledge no published research has tackled the problem of using program analysis to reduce the traffic overhead of serialization. Thus, no other approach is directly comparable to ours.

Krishanswamy et al. [8] reduce the computational overhead on the client side by exploiting object caching. For each call, a copy of the byte array storing the serialized object is cached to be possibly reused in later calls. Braux [1] exploits static analysis to reduce the computational overhead of an invocation due to the reflective calls needed to discover the dynamic type information. The work of Kono and Masuda [7] relies on the existence of run-time knowledge about the receiver's platform, and redefines the serialization routine accordingly. On the sender side, the object to be serialized is converted directly into the receiver's in-memory representation, so that the receiver can access it immediately without any data copy and conversion. Breg and Polychronopoulos [2] explicitly target homogeneous cluster architectures, and provide a native implementation of a subset of the serialization protocol. Their approach leverages on knowledge about the data layout in the cluster, so that complex data structures are encoded directly in the byte stream by using only a minimal amount of control information. Philippsen et al. [12] integrate various approaches to obtain a slightly more efficient RMI implementation. They simplify the type information encoded in the serialization stream, improve the buffering strategies for dealing with the stream, and introduce a special handling of `float` and `double`. Nevertheless, their optimizations are again closely tied to the parallel computing domain.

8 Conclusions

We presented a novel program analysis technique that aims at optimizing parameter serialization in remote method invocations on a per-invocation basis.

The analysis identifies which portion of a parameter is actually used on the receiving side. This information can be exploited to redefine the serialization mechanism and reduce the run-time communication overhead. We implemented a toolkit supporting our approach, and we are currently using it for an empirical evaluation of our method against real-world RMI applications.

Acknowledgments The work described in this paper was partially supported by the Italian Ministry of Education, University, and Research (MIUR) under the VICOM project, and by the National Research Council (CNR) under the IS-MANET project.

References

1. M. Braux. Speeding up the Java Serialization Framework through Partial Evaluation. In *Proc. of the Workshop on Object technology for Product-line Architectures*, 1999.
2. F. Breg and C.D. Polychronopoulos. Java Virtual Machine Support for Object Serialization. *Concurrency and Computation: Practice and Experience*, 2001.
3. J.-D. Choi et al. Escape Analysis for Java. In *Proc. of OOPSLA'99*, pages 1–19. ACM Press, 1999.
4. C. Ghezzi, V. Martena, and G.P. Picco. Enhancing Remote Method Invocation through Type-Based Static Analysis. Technical report, Politecnico di Milano, 2003. Available at `www.elet.polimi.it/~picco`.
5. Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `www.cc.gatech.edu/aristotle/Tools/jaba.html`.
6. K. Kennedy. A Survey of Data Flow Analysis Techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.
7. K. Kono and T. Masuda. Efficient RMI: Dynamic Specialization of Object Serialization. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 308–315, 2000.
8. V. Krishnaswamy et al. Efficient Implementations of Java Remote Method Invocation. In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, pages 19–36, 1998.
9. W. Landi and B. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.
10. Sun Microsystems. Java Remote Method Invocation Specification, 2002.
11. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.
12. M. Philippsen et al. More Efficient Serialization and RMI for Java. *Concurrency—Practice and Experience*, 12(7):495–518, 2000.
13. S. Sinha and M.J. Harrold. Analysis of Programs That Contain Exception-Handling Constructs. In *Proc. of Int. Conf. on Software Maintenance*, pages 348–357, Nov. 1998.
14. B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of the Symp. on Principles of Programming Languages*, pages 32–41, 1996.
15. M. Suedholt and C. Steigner. On Interprocedural Data Flow Analysis for Object-Oriented Languages. In *Proc. of the Int. Conf. on Compiler Construction*, LNCS 641, 1992.