The following paper was originally published in the
Proceedings of the USENIX Conference on Object-Oriented Technologies (COOTS)
Monterey, California, June 1995.

Phantom: An Interpreted Language for Distributed Programming

Antony Courtney
Department of Computer Science
Trinity College Dublin, Ireland

Phantom : An Interpreted Language for Distributed Programming

Antony Courtney*
Department of Computer Science
Trinity College Dublin
Ireland

Abstract

The emerging trend in writing distributed applications is to use an object-based RPC system with a statically compiled, object-oriented language. While such a programming environment is adequate for many tasks, object-based RPC systems and statically compiled languages also have certain intrinsic limitations. These limitations become significant when writing applications which are both distributed and interactive (e.g. network information browsers, distributed conferencing systems and collaborative work tools). This paper discusses these limitations, and presents the design of *Phantom*, a new intepreted language for distributed programming. Phantom provides many features found in object-based RPC systems and statically compiled languages, including *automatic marshalling*, *transparent remote procedure call*, *secure authentication* and *concurrency support*. In addition to these traditional features, Phantom's interpreted nature permits the use of certain programming techniques, such as *true object migration*, *remote evaluation*, and *dynamic extensibility*, which are of increasing importance for distributed programming, but which are not available in statically compiled languages and RPC systems. The integration of these features in a single, coherent programming language makes whole new classes of distributed, interactive applications possible.

1 Motivation

1.1 Object-based RPC and Static Compilation

The current trend in programming distributed systems is to use a statically compiled, object-oriented language (such as C++ [Str92]), augmented with an object-based RPC system and associated runtime library (such as CORBA [Gro92] or ILU [JSS94]).

With an object-based RPC system, the programmer specifies the interface to network-accesible objects using an *interface definition language* (IDL). A *protocol compiler* compiles the IDL specification to generate client and server *stub routines* which are linked with the programmer's application code. For each remote object which the application has access to, the

*email: Antony.Courtney@cs.tcd.ie

client stub routines provide a *local surrogate object*. The local surrogate object appears (to the application programmer) like a normal programming language object. However, when the application invokes a method on the local surrogate, the surrogate sends an RPC to the remote server object which actually performs the request, and the local surrogate returns the result of the RPC as if the operation were performed locally. This provides a degree of *location transparency* for the programmer: there is no way of distinguishing method invocation on true objects from method invocation on surrogate objects.

Using a statically compiled, object-oriented language with an object-based RPC system provides a number of benefits over unstructured, transport-layer message passing. In particular, such systems solve heterogeneity problems, provide location transparency, and add structure to the network communication between different programs.

However, object-based RPC systems and statically compiled languages also have some significant deficiencies. For purposes of this paper, the most significant deficiency is that object-based RPC systems do not provide any mechanism for sending code across sites. This excludes certain distributed programming techniques (such as Remote Evaluation [SG90]) and, as the following example will illustrate, also places severe limitations on both the dynamic extensibility and performance of interactive applications.

1.2 An Example from the World-Wide Web

The limitations of object-based RPC systems and statically compiled languages become significant when writing applications which support both remote network access and interactive user interfaces. An illustrative example of these limitations comes from the World-Wide Web (WWW) [BLCGP92].

A typical example of an interactive application for WWW is Lawrence Berkeley Laboratory's "Interactive Frog Dissection Kit" [RJN94], an educational program which uses three dimensional volume rendering to allow a user to graphically explore the anatomical structures of a virtual frog. A screen-shot of the program is shown in figure 1.

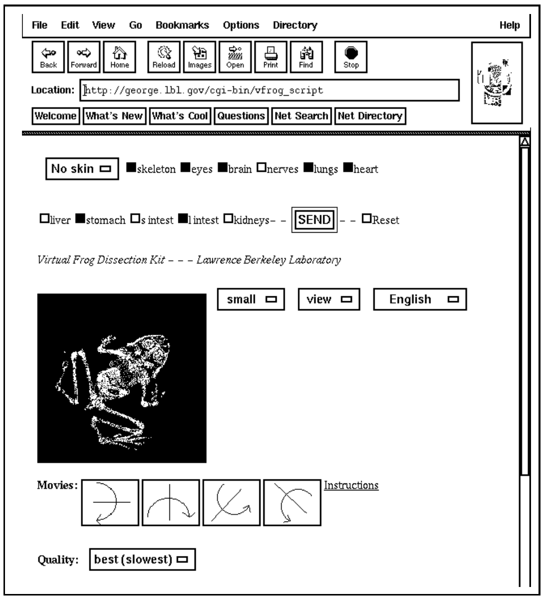

Figure 1: An interactive WWW application.

This application has two parts. There is a *client* part, which displays the visual image of the current view of the frog, and allows the user to change this view by setting various parameters. In the current implementation, the client part is an HTML [BLC93] form, which is displayed and managed by the user's WWW browser program. There is also a *server* part, which runs on a remote, high-performance computer and renders the three-dimensional image of the current view of the frog. In the current implementation, the server is a C program run via WWW's Common Gateway Interface (CGI).

An alternative approach to using HTML forms and CGI for this application would be to use object-based RPC between the interactive client program and the remote rendering server. There are two possibile approaches to doing this:

1. Develop special-purpose client software to access the server, specifically for this application. This gives the programmer the freedom to decide what to compute on the client side, what to compute on the server side, and how much state to keep between the client and the server. The programmer can apply specialised knowledge of the application to optimise interactive response in the client, and minimise network usage between the client and server. However, this approach requires that users obtain and compile the special-purpose client software. This added burden on potential users precludes exactly the sort of casual exploration which has made WWW (and its many interactive applications) so popular.

2. Develop a completely general client program, which can be used for many different interactive applications and which allows remote server programs to create and manage arbitrary GUI objects in a client window. The problem with this approach is that if the client program is sufficiently general, giving the server arbitrary control over the client window, then all responses to interactive events must be handled by the server application. Experience with the X Window System (and its various object-based toolkits) has shown that remote handling of interactive events simply will not yield acceptable response time over networks with higher latencies than a high-speed LAN.

The use of CGI and HTML for such applications has two important benefits over using object-based RPC between the client and the server. First, any WWW browser which supports forms can serve as a client program for any specialised application server. Second, the browser program handles all direct user interaction with the HTML form for the server. This ensures that the client user interface will have adequate interactive response for those GUI events (such as individual key presses) which do not require server interaction.

However, using CGI and HTML to build interactive applications creates its own set of problems:

- The set of interaction mechanisms provided by HTML forms, and their presentation by the WWW browser, is relatively fixed. For example, HTML does not currently provide any mechanism for creating dialog boxes or handling modal interactions.

- The benefit of local interactive response is limited to those applications which fit cleanly into a "forms" model of interaction, such as database clients. Applications which don't fit cleanly into the forms model (i.e. because they require application-specific behaviour for each interactive event) are either impossible to develop or suffer from exceedingly poor interactive response. For example, it would be impossible to develop a soft real-time game using HTML and CGI.

- Items in the GUI presented by a CGI program can not be changed individually; the CGI program must send a new form back to the client for every minor change in the interface's appearance.

- There is no server-side state maintained; the client must re-transmit the entire form contents to effect any change in the server.

The problem, effectively, is that a general-purpose document specification language (HTML) is being used as a batch-oriented programming language for specifying and controlling Graphical User Interfaces.

We believe that there is an alternative approach to implementing interactive, distributed applications (such as the frog dissection kit), which does not sacrifice generality or interactive response time. Instead of a remote application server transmitting an HTML document to the local browser, our approach is to have the remote application transmit a procedure (written in a general purpose programming language) across the network, which the browser will execute. This procedure will be given complete control over some region of the user's display (e.g. a window), so that the programmer of the application server is not tied to any fixed set of user interface mechanisms. And because the procedure received from the application server is executed locally, interactive response time will not suffer.

1.3 The Phantom Approach

To address the problems inherent in writing applications which support both remote network access and interactive user interfaces, we have developed *Phantom*, a new language and runtime environment for writing distributed applications[1]. The goal of Phantom is to provide a single, powerful infrastructure for developing large-scale, interactive, distributed applications. To meet this goal, Phantom attempts to redress some of the deficiencies of statically compiled languages and RPC systems. Phantom is able to address these deficiencies for the following reasons:

1. *Program code may be transmitted across sites.* This is, perhaps, Phantom's single biggest advantage over statically compiled languages and RPC systems. Phantom provides generalised concepts of *higher-order functions* and *lexical scoping* in the context of a distributed system. The use of higher-order functions provides a semantically sound, easily-understood mechanism for sending small parts of programs (individual procedures and functions) to remote sites for execution. Lexical scoping is used to guarantee that procedures received from remote sites will not have access to any resources or information on the local site which could not be accessed via RPC from the remote site.

2. *The language was designed with distribution in mind.* The Phantom language appears to application programmers much like a conventional programming language. However, certain details of the type system (such as the elimination of arbitrary pointer types) make it possible for the runtime to provide transparent distribution facilities for Phantom code and data.

3. *The language is interpreted.* Using an interpreter for the language makes the run-time environment very flexible. In particular, implementing Phantom as an interpreter makes it easy to support very high-level data types, perform automatic storage management, provide general-purpose higher-order functions, dynamically load and execute Phantom programs, and provide transparent access to code and data at remote sites.

[1] The name *Phantom* derives from the "transparent" nature of the language's distributed programming features.

4. *Distribution is flexible and transparent.* In statically compiled languages and RPC systems, extra programming work is required to specify which values or procedures may be made accessible across the network, and the programmer is burdened with even more work if the data types to be shared are linked or cyclic. In Phantom, all program values are potentially and transparently network accessible. However, the details of distribution are kept under flexible program control, so that programmers may optimise their applications for performance.

While none of the above features is compelling in itself, their integration in a single, simple language provides a powerful environment for constructing distributed applications.

2 Related Work

Distributed programming languages are not a new idea. An excellent overview of other distributed programming languages is given in [BST89], which includes a bibliography of over 200 papers on nearly 100 different languages. Despite the number and variety of these other distributed programming languages, most other distributed languages are intended for harnessing the power of a network of processors as a single parallel processing engine. Furthermore, most other distributed programming languages do not provide any mechanism for transmission of code across sites. The only language we are aware of which provides facilities for transparent distribution of both program code and data is Obliq [Car95].

The distribution model of Phantom uses the same basic model as Obliq. However, there are a number of aspects of Phantom which differentiate it from Obliq:

- Phantom is strongly typed (using structural type equivalence). The adherence to strong typing provides a level of static error checking which is not provided in most other interpreted languages (including Obliq, Tcl [Ous94] and Python [Ros92]). As examples will illustrate, Phantom provides *type-safe implicit declarations*, which limit the overhead of strong typing in an interpreted language.

- Phantom uses a class-based object model, rather than prototypes. This, in conjunction with strong typing, encourages the seperation of interface from implementation of object types.

- Syntactically, Phantom more closely resembles Modula-3.

- Phantom provides a simple access control and authentication mechanism at the language level.

- The Phantom interpreter is implemented purely in ANSI C, and provides a library interface to the Tk toolkit [Ous94].

Although we chose to design a new language, producing a novel new programming language is not the goal of the project. We tried hard to borrow proven mechanisms from other languages and systems rather than invent our own. The language core is based on the syntax and semantics of Modula-3, so that it will be accessible to application programmers with exposure to any successor of Pascal. For distribution, Phantom uses the distributed lexical scoping semantics of Obliq. For access control, Phantom uses a model which is similar to permissions in the UNIX file system [RT74].

3 The Phantom Language: Concepts and Techniques

The Phantom language supports a number of modern imperative programming features, including: *interfaces*, *objects*, *threads*, *garbage collection* and *exceptions*. These are all features of the programming language Modula-3 [Nel91]. Where possible, Phantom borrows Modula-3's syntax and semantics. Phantom extends Modula-3's object model (by adding the concepts of *ownership* and *access control*), and omits many of Modula-3's more complex features (such as *reference types* and *generics*) which would complicate Phantom's distributed semantics. Phantom also includes support for implicit declarations, dynamically-sized lists, and general purpose higher-order functions. These features are modelled on their counterparts in other interpreted languages (such as Scheme [IEE91] and Python [Ros92]).

3.1 Object Model

Phantom supports object-oriented programming, in a manner similar to Modula-3. Phantom objects have *attributes* (containing state information), a number of *methods* (for performing operations), and support single-inheritance. Phantom uses a class-based object model (rather than prototypes, as in Obliq [Car95] or Self [US87]). While classes are more verbose than prototypes, we feel that classes provide for a cleaner separation between the interface and implementation of objects, and scale better for large applications.

Objects are the focus of communication in Phantom. A Phantom program will generally make its services available to other programs by registering object values with a *name service* – a Phantom application server which maps string names to network addresses[2] of Phantom objects.

Objects (and lists, which are a kind of object) are the only values which are passed by reference. This has an important property when passing object values to Phantom programs at remote sites: objects are never implicitly migrated to remote sites as the result of an assignment, procedure call or return statement. Instead, the object remains stationary and a network reference is passed to the remote site. If migration of objects across sites is required, it must be performed explicitly by the programmer. While this violates location transparency to some degree, we feel that only the programmer can make reasonable decisions about when and where to migrate objects.

3.2 Distribution Model

The distribution model of Phantom borrows heavily from the distributed lexical scoping semantics of Obliq [Car95]. The basic concepts are illustrated in figure 2.

[2]These network addresses take the form of *global location addresses*, described in section 3.2.

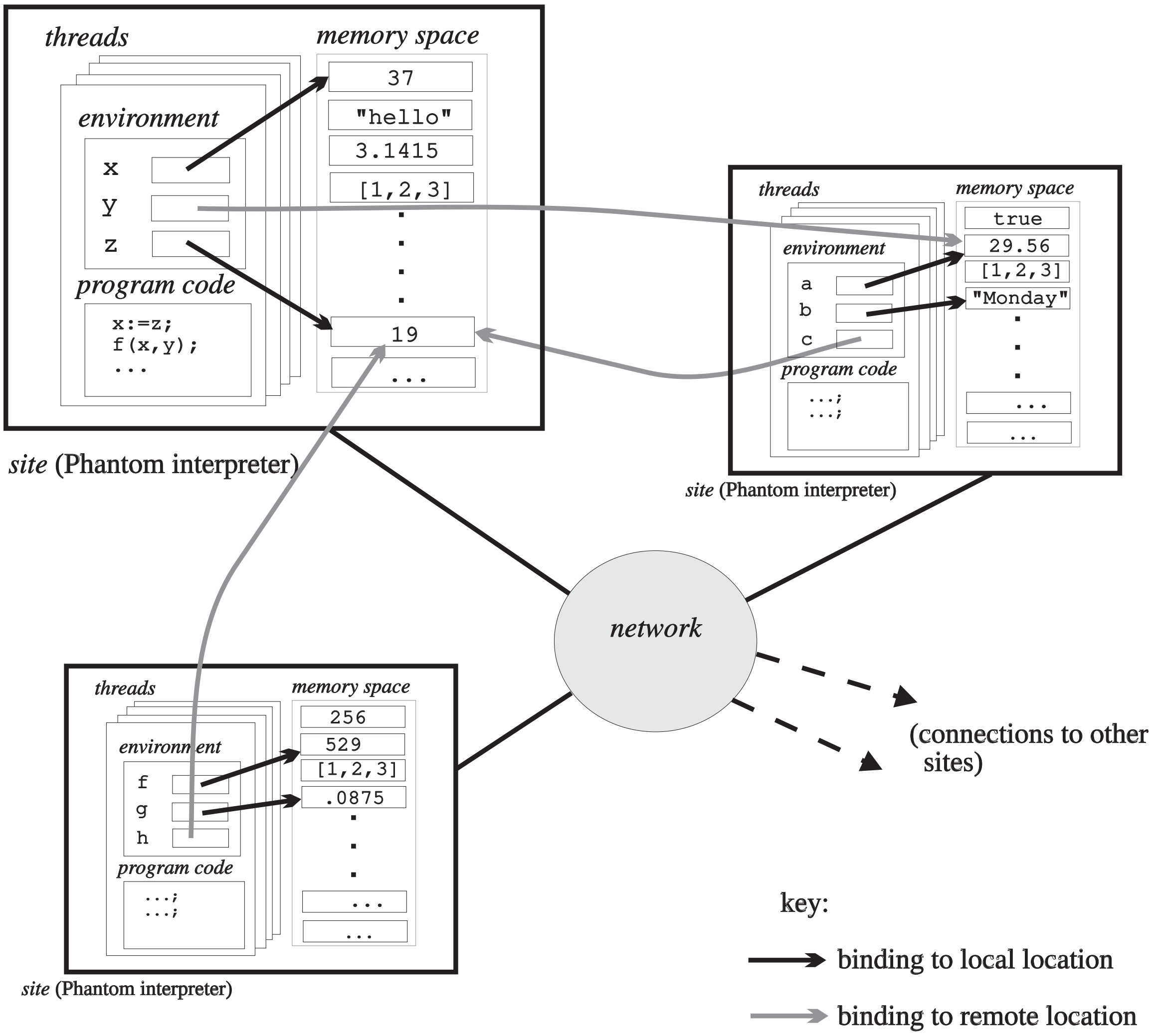

Figure 2: Phantom network architecture.

A *network* connects a number of *sites*. A *site* is an invocation of the Phantom interpreter on some host machine, and has a *site address* which uniquely identifies that site throughout the network. In the current implementation, a site address is simply a pair consisting of the IP address and port number of a TCP socket owned by the interpreter process. Note that a host running a multi-tasking operating systems may contain several sites (corresponding to multiple invocations of the Phantom interpreter).

Within a site, the interpreter maintains a single *memory space* for the program it is executing. This memory space contains a number of *locations*. Each location has a *location address*, which uniquely identifies the location within the interpreter's memory space, and holds a *value*.

Each Phantom program executes as a number of *threads* within the interpreter. Threads are provided through a library which implements the POSIX pthreads specification [IEE92]. Two data structures are associated with each thread:

1. A representation of the program code which the thread is executing. In the current implementation, the interpreter uses a sequence of byte codes for a virtual stack machine for this purpose.

2. An *environment*, which maps every variable or constant identifier appearing in the Phantom program to a *global location address*. A *global location address* is a pair consisting of a site address and a location address within the memory space of that site.

The Phantom interpreter uses environments to provide transparent distribution for Phantom programs. Each statement in a Phantom program may make reference to constant and variable identifiers. As the interpreter executes a statement, it uses the program's current environment to map these identifiers to their corresponding global location addresses. The interpreter then performs the appropriate operation on each location according to the defined semantics of the language. If the global location address refers to a location within the local interpreter's memory space, the operation is performed directly by the interpreter. If, however, the global location address refers to a location in the memory space of another site, the interpreter sends a request to the remote site asking it to perform the given operation.

3.3 Example Application: Generic Client and "Hello" Server

The following example illustrates how the interpreter and runtime provide Phantom's distributed semantics, and also illustrate the basic techniques for developing dynamically extsenible, distributed, interactive applications in Phantom. There are two programs presented. The first is a generic client program (such as might be launched as an "external viewer" from a WWW browser), and the second is a specific application server with which the client can communicate.

The generic client uses the information in a Uniform Resource Locator (URL) [BL94] to obtain a reference to a remote application server. Once the client has obtained a reference to the application server, it obtains an autonomous *agent* from the sever: a procedure received from the remote site which the client executes locally.

The "Hello" Server is as an example of a specific application server. It accepts requests from clients and returns to each client an agent which creates an instance of an interactive "hello world" object at the client's site.

The general purpose client could be invoked using a command line such as:

```
$ phi AppClient phi://server.host.name/HelloServer
```

which starts the Phantom interpreter (`phi`) executing the module `AppClient` (the name of the client program) with the URL for the Hello server as a command-line argument available to the client.

The agent obtained from the Hello server creates a window on the client's display as shown in figure 3:

Figure 3: Window Created on Client's Display by Hello Server

All user interface events for this window are handled by code for the agent, which is executed at the client site. When the user presses the button labelled "Hello", the agent responds by printing the message `''Hello, World''` to its output stream. When the user presses the button labelled "Quit", the agent returns control to the client program, which then exits.

While this is a simple example, it serves to illustrate most of the important features of the application domain for which Phantom was designed. This example is both distributed and interactive, the client is dynamically extensible, and all interactive user interface events are handled at the client site. These same principles apply to other, more sophisticated applications in the target domain.

3.3.1 Application Server Interface

An application server makes its services available to clients by registering an instance of an `AppServer.T` object with a name server. The `AppServer.T` type is implemented by *every* specific application server, and is also known to the generic client. This shared interface is as follows:

```
interface AppServer;

import rd, wr;

(* An Agent is simply a procedure executed at the client site *)
type Agent = proc (istrm: rd.T; ostrm: wr.T);

(* AppServer.T -- an application server *)
type T=object (serialised, protected)
methods
  (* generate a new agent for execution at the client *)
  generate_agent(): Agent perm x;
end;

end AppServer.
```

This interface defines two types: `AppServer.Agent` (describing the type of the agent given to the client for local execution), and `AppServer.T`, the abstract type of an application server. The generic client obtains an agent from an application server by first obtaining a reference to an `AppServer.T` (using the name service), and then invoking its `generate_agent()` method. As will be shown later, specific application servers are implemented by creating subtypes of `AppServer.T`.

The type `Agent` makes use of the fact that procedures are first-class types in Phantom: procedures may be assigned to variables, passed as parameters, and returned from procedures, just like values of other fundamental language types. In this example, an `Agent` is a procedure taking two parameters (which must be supplied by the client). Such parameters represent the *services* which an *execution site* (i.e. the client) makes available to agents it receives from across the network. In a more general interface, such services would encapsulate all of the local resources the execution site is willing to provide to the agent: a local audio service, a 3-D rendering service, etc. For simplicity, the only services provided to agents in this example are input and output streams for reading and writing messages.

The object type, `T`, has two *qualifiers* (the `serialised` and `protected` keyword qualifiers[3]), no attributes, and a single method: `generate_agent()`.

[3] For those familiar with Obliq, both of the `serialised` and `protected` qualifiers have the same semantics as their counterparts in Obliq.

The *qualifiers* give the object type special semantics. A `serialised` object type ensures that only one external method invocation or attribute update is active at a time in the presence of multiple concurrent requests. The `protected` qualifier prevents any attributes of the object from being updated externally, and prevents the object from being copied. The `protected` qualifier is of little immediate use on the object type defined here (since it has no attributes), but the qualifier is part of the type, and will hence be carried down into subtypes of `T`.

Ownership and Access Control

The `generate_agent()` method of type `T` has a name, a procedure signature, and a *permissions specification* (given by `perm x` following the signature). Permissions specifications are used to set the *access control* properties of attributes and methods.

Each Phantom object is stored in the memory space of an interpreter, which communicates with other interpreters across a network. Each object has an *owner*, which is represented at runtime as a `sys.user` object corresponding to the user who started the interpreter containing the object.

The permissions specification specifies what operations on the object may be performed by users other than the owner. An operation on an object by a user other than the owner happens when the interpreter receives a request from a Phantom program running on a different site, and is discussed in detail in section 3.5.2.

Each permissions specification consists of a bit-mask of zero or more of the three *permission bits* **r**, **w** and **x**, corresponding to *read*, *write* and *execute* permission, respectively.[4] If no permissions specification is given for an attribute or method, the default is that all permission bits are turned off.

In the current example, the `generate_agent()` method of `AppServer.T` has its `x` bit set in the permissions specification to allow clients at remote sites to invoke this method.

3.3.2 Client Program

The client program is straightforward: it obtains the global location address of an `AppServer.T` object from the name service (defined in the interface `ns`) using the application's URL, invokes the `generate_agent()` method of the server to obtain an agent, and executes the agent locally, supplying the standard input and output streams of the client as the parameters to the agent. The code for the client is as follows:

```
(* AppClient.pm -- implementation of a general-purpose network client *)
module AppClient;

import AppServer, Tk, stdio, ns, sys, urllib;

const
  urlstring = "phi://server.host.name/HelloServer";

begin
  try
    url:=urllib.parse(urlstring);
    name_server:=ns.find(url.host);
```

[4] Note that the **r** and **w** bits apply only to attributes, and the **x** bit applies only to methods.

```
    app_ref:=name_server.lookup(url.path);
    app_server:=narrow(app_ref, AppServer.T);

    (* obtain the agent from the server *)
    local_agent:=app_server.generate_agent();
    (* and execute the agent locally *)
    local_agent(stdio.stdin,stdio.stdout);
  except
    urllib.malformed =>
         stdio.stderr.puts("error: malformed URL: " @ urlstring @ "\n");
  | sys.narrow_failure =>
         stdio.stderr.puts("error: URL does not refer to an AppServer\n");
  | ns.not_available =>
          stdio.stderr.puts("error: could not contact name server at host " @
                            url.host @ "\n");
  | ns.unknown_service =>
          stdio.stderr.puts("error: application " @ url.path @
                            " not registered with nameserver.\n");
  end;

end AppClient.
```

The first item to note in the above example is that there are no explicit variable declarations. In Phantom, it is not necessary to declare a local variable prior to its use in a statement block. The first assignment to an undeclared identifier will declare that variable in the local scope, with a type derived from the expression on the right-hand side of the assignment statement[5]. All subsequent references to the identifier in the statement block will be type-checked against this automatically derived type.

The client program works as follows: First, the program calls `urllib.parse()` to parse the URL given by `urlstring`. (In this example, `urlstring` is given as a constant; in practice it would use a command-line argument.) The procedure `urllib.parse()` returns the URL as a record with seperate `protocol`, `host` and `path` fields. Next, the client attempts to contact a Phantom name sever running on the host given in the URL, using the procedure `ns.find()`. The `ns` module and interface is part of the standard Phantom library, and locates a name server object on a local or remote site using a well known TCP port. The variable identifer `name_server` is assigned the global location address of the name server object, returned from the call to `ns.find()`. Any subsequent operation on the identifier `name_server` is forwarded transparently by the interpreter to the name server object, which performs the operation and returns the result.

Next, the `lookup()` method is invoked on the `name_server` object to obtain a reference to the application server, using the pathname part of the URL (`"HelloServer"` in this example). The `lookup()` method returns its result as type `any`; the statement following the lookup performs a type-safe runtime type conversion using `narrow()` to convert this value to an object reference of the appropriate type. Note that after the client performs the `lookup()` operation, the name server is no longer involved in the communication between the client and the server; it just

[5] Note that explicit variable declarations may still be provided (since they increase readability for large programs), in which case the type given in the explicit declaration is used for type checking.

provides a mechanism for bootstrapping the connection between them. Once the client has a reference to the remote `AppServer.T` object, operations can be performed on the object reference in the same manner as with the `name_server` object; the runtime handles any network communication required.

Next, the client invokes the `generate_agent()` method of the `app_server` to obtain an agent for local execution. Since the value returned from `generate_agent()` is a procedure, this will result in obtaining the code for the procedure from the application server. This code will be dynamically loaded into the memory space of the client, and the variable `local_agent` will refer to the *closure* for this procedure. As will be discussed later, the semantics of transmitting code across sites ensures that this is a safe operation: code received from a remote site and executed locally can not gain unauthorised access to any local resources.

Finally, the client invokes `local_agent`, passing as parameters the standard input and output streams of the client program. Thus, the client has no information about specific applications hard coded into it, but dynamically obtains application-specific behaviour by receiving code from the server. This generic client program could be used without modification as a client for any application-specific server.

It is also worth noting that the client wraps the entire body of its mainline in a `try`-`except` statement, to catch some of the exceptions which may be raised in the process of obtaining the application-specific agent, and reports these as errors to the user. More sophisticated error recovery mechanisms could be implemented using this facility.

3.3.3 Server Program

The application-specific server is a "Hello, World" server. It returns to clients an agent which, when executed at the client site, creates a graphical, interactive "Hello, World" window on the client's display. The agent uses the library interface between Phantom and the Tk toolkit to implement the graphical user interface for the agent.

The server defines the type `ServerImpl` as a subtype of `AppServer.T`. This is a common technique in Phantom: an object type appearing in an interface will describe the external view presented to clients, and a subtype will be used to implement the application-specific server. The source code for the server program is as follows:

```
module HelloServer;

import AppServer, Tk, rd, wr, ns, stdio;

(* ServerImpl is the type of the "hello" application server; implemented as a
 * subtype of AppServer.T
 *)
type ServerImpl=AppServer.T object
end;

(* Hello is the object type instantiated at the client site *)
type Hello=Tk.Frame object
   quit: Tk.Button;
   msg: Tk.Button;
   wstrm: wr.T;     (* stream on which to write messages *)
methods
```

```
   CreateWidgets();
   say_hi();
end;

(* methods of Hello: *)
proc Hello.CreateWidgets(self: Hello)
begin
  self.quit:=new(Tk.Button,
                 master:=self,
                 text:="Quit",
                 fg:="red",
                 command:=lambda () { self.exit(); });
  self.quit.pack(side:=Tk.left);
  self.msg:=new(Tk.Button,
                master:=self,
                text:="Hello",
                command:=lambda () { self.say_hi(); });
   self.msg.pack(side:=Tk.left);
end;

(* init() method -- called automatically to initialise new instances *)
proc Hello.init(self: Hello)
begin
  Tk.Frame.init(self); (* call super-class init method *)
  self.pack();
  self.CreateWidgets();
end;

proc Hello.say_hi(self: Hello)
begin
  self.wstrm.puts("Hello, world!\n");
end;

(* methods of Hello Server: *)
proc ServerImpl.generate_agent(self: ServerImpl): AppServer.Agent
 (* client_agent() is the procedure returned by generate_agent() and
  * executed at the client site
  *)
 proc client_agent(istrm: rd.T; ostrm: wr.T)
 begin
   hello_app:=new(Hello,
                  wstrm:=ostrm);
   hello_app.main_loop();
 end;
begin
  return client_agent;
end;
```

```
begin
  (* create an instance of the server, and register it with the local name
   * service
   *)
  hello_server:=new(ServerImpl);
  name_server:=ns.find();
  name_server.register("HelloServer",hello_server);
end HelloServer.
```

The server is implemented as follows:

First, the actual server object is implemented as a subtype of the object type `AppServer.T`. The subtype (`ServerImpl`) does not add any attributes or methods to `AppServer.T`, it simply overrides the `generate_agent()` method of `AppServer.T`. Hence, the body of the `ServerImpl` is empty, since it does not have any specific attributes or methods, and, in Phantom, method overrides are not stated explicitly in the object type.

Next, the application server defines the object type `Hello` as a subtype of `Tk.Frame`. No instance of this type is ever created at the server site; instead, an instance of this type is created at the client site by the application-specific agent. When the agent is transmitted from the server to the client, all information about types used within the agent is transmitted across the network and reconstructed at the client site. For object types, this includes both the information necessary to construct instances of the type, and the code for any methods. Note that a type may refer to other types in its definition, and types may be recursive; the runtime will transmit all necessary type information, including information about types referenced indirectly or recursively.

The agent returned to clients is the procedure `client_agent()` defined in the `generate_agent()` method of `ServerImpl`. The `client_agent()` procedure (executed at the client site) creates a new instance of type `Hello` at the client site, and invokes the `main_loop()` method of `Hello` to process GUI events which happen in this object. The `main_loop()` method of `Hello` is inherited from `Tk.Frame`, the parent type of `Hello`.

Finally, the mainline of `HelloServer` creates a new instance of `ServerImpl`, and registers this with the local name server. When the Phantom interpreter is invoked to run the server application, it would be invoked with the `-noexit` option, to ensure that the interpreter does not exit after initialisation, but instead waits idly for requests from remote sites.

3.4 Semantics of Procedure Transmission

The previous example makes use of the fact that procedures are first class types in Phantom. This has an important property when passing parameters to (or returning values from) methods of remote objects. If a procedure value is passed as a parameter or returned as a return value, a *closure* for the procedure is transmitted across the network.

Phantom programs are lexically scoped: that is, the location to which a free identifier is bound is purely a static function of where the procedure is defined, and is not a dynamic function of the procedure call stack. This property is used to give higher-order procedures an intuitive and secure meaning in a distributed context. These semantics are illustrated in figure 4:

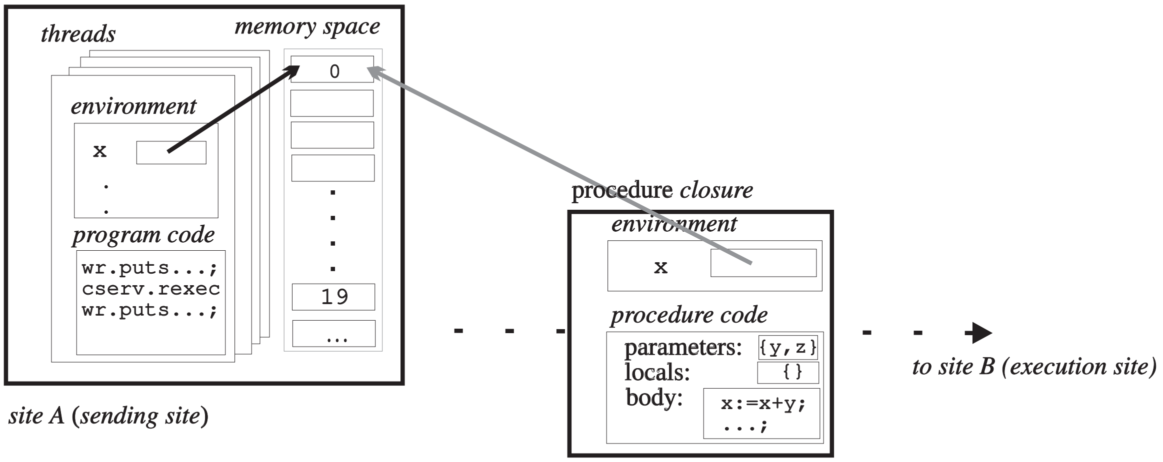

Figure 4: Transmission of closures across sites.

When a procedure value is sent to a remote site, the local interpreter sends the procedure as a *closure*. The closure contains two pieces of information:

1. A representation of the code in the body of the procedure. This could be either a direct representation as source code text, or some internal representation such as a linearised parse-tree or byte-codes for a virtual machine.

2. An *environment*, which maps all free variable identifiers which appear in the procedure to global location addresses.

Transmitting the set of bindings along with the code for the procedure preserves the correct lexical scoping semantics when the procedure is executed at the remote site. When the procedure body makes reference to a free identifier, the binding to the global location address ensures that the operation is performed on the location where the identifier was bound originally.

The "Hello, world" example illustrates a limited case of transmitting procedures across sites. In that example, the procedure which is transmitted to the client site as the application-specific agent has no free variable identifiers. That is, the procedure `client_agent()` does not refer to any variables from its enclosing scope. This is an example of a "disconnected" agent: all information needed to execute the procedure at the client site can be encapsulated in the code of the closure, and the closure's environment will be empty. Although `client_agent()` does refer to *types* (such as the `Hello` object type[6]) from enclosing scopes, this type information is transmitted to the client site as part of the code of `client_agent()`'s closure.

If the body of `client_agent()` made reference to a variable in its surrounding scope, the environment of the closure transmitted to the client would contain a binding to the location of the variable at the server site. This would have the effect of creating a "connected" agent: one which carries its network connections with it. This facility could be used, for example, to create a distributed multiplayer game. The agent transmitted to the client could simply invoke operations on an object (representing the opposing player) declared in one of the agent's enclosing scopes. Any time the agent (executing at the client site) performed such an operation, the

[6]Note that the methods of object types are themselves procedures. When the information about an object type is sent across sites, the methods of the object type are transmitted as true closures in the same way as other procedures.

client runtime would use the environment transmitted with the agent to forward the operation to the site where the object resides.

3.5 Security Considerations

Phantom's distribution model raises a number of interesting security issues. The most important issue is that raised by the ability to send code across sites: the language and runtime must provide strong guarantees about the safety of executing code received from a potentially untrustworthy server. Phantom addresses this issue, and also provides two other forms of security support: secure user authentication and access control, and unforgeable global location addresses.

3.5.1 Security of Code Transmission

The principle concern when transmitting code across sites is security. The language and runtime environment must be able to guarantee that program code which is received from a remote site and executed locally will not have access to any local resources which could not have been accessed via RPC from the remote site.

Phantom makes this guarantee through adherence to lexical scoping in the context of distribution and higher-order functions. In practical terms, the implementation guarantees lexical scoping by passing a set of bindings for all free identifiers along with the code for a procedure. When an interpreter receives a procedure from a remote site, it can perform a single, static check to ensure that all free identifiers in the code for the procedure have a corresponding entry in the set of bindings received with the procedure. If any free identifier does not have a corresponding binding, the interpreter will abort the operation requested by the remote site and return a security violation exception.

The language has no general purpose pointer types, which is a crucial aspect of this approach to security. Eliminating general purpose pointers ensures that the only way for a procedure to refer to resources outside the body of the procedure is through free identifiers. Because the implementation ensures that free identifiers are handled through strict lexical scoping, executing procedures received from remote sites is guaranteed to be secure: there is simply no mechanism for the procedure to gain unauthorised access to any local resources.

3.5.2 Authentication and User Identity

In Phantom, the identity of every user has a runtime representation as a `sys.user` object. Procedures in the standard interface `sys` provide access to a `sys.user` object which represents the *current effective user indentity.* A simple, secure authentication protocol (based on public key encryption) is used when the connection between two Phantom interpreters is first established. The secure authentication protocol ensures that an interpreter can obtain an unforgeable `sys.user` object representing the user who started execution of the interpreter at the remote site. This object is used as the current effective user identity when the local interpreter executes any methods or procedures as the result of a request from the remote interpreter.

As discussed in section 3.1, every attribute or method of an object has a set of access control bits which determine the operations that may be performed on the object by remote users. Access checks performed by the interpreter are always based on the current effective user identity. An object operation will succeed if either the user running the current thread is the owner of the object (which is always true for purely local operations), or if the permission bits permit relevant access to the attribute or method for other users.

The security model provided at the *language level* by Phantom is minimal by design. The goal of the language-level primitives is to provide a simple, secure mechanism for partitioning an application into parts which are and are not network-accessible. For large-scale applications, which have more sophisticated security requirements, we plan to augment the language-level primitives with a library that provides complex principals, access control lists and encryption, in a manner similar to Taos [WABL93].

3.5.3 Global Location Addresses and Security

A global location address is the handle used by an interpreter to access locations stored in the memory space of remote sites. As described in section 3.2, a global location address consists of a site address (of the interpreter which owns the location), and a location address (relative to the memory space of that site).

The relative location address part of a global location address is a 128-bit key generated by the site which owns the location, rather than just a pointer into the interpreter's memory space. Because this key space is large and sparsely populated, the key acts as a software *capability* [Fab74] for locations in the interpreter's memory space. Thus, private data can be safely stored in ordinary Phantom program variables and shared with authorised users, without fear of a rogue site gaining access to the data by guessing the location's address.

4 Current Status

At the time of writing, the design of the language and interpreter are complete. The interpreter has been implemented, and supports all of the features in the language core, including static typing, type-safe implicit declarations, objects, interfaces, threads, exceptions, garbage collection, dynamically sized lists and higher-order functions, and includes a library interface to the Tk toolkit. A number of small demonstration programs have been written in Phantom. The initial results are limited but encouraging: the language's Modula-3 heritage affords it a number of powerful features, while still maintaining overall coherence and simplicity.

The implementation of the networking subsystem in the interpreter required for distribution is not yet complete; we expect to complete that part of the interpreter over Summer, 1995.

5 Future Work

Once the distribution facilities in the interpreter are available, we intend to use Phantom to develop a number of novel distributed applications. The most compelling of these ideas is a graphical, extensible distributed conferencing system. This will involve implementing a distributed version of LambdaMOO [Cur92] in Phantom. LambdaMOO is a "text-based virtual reality" which uses its own interpreted, object-oriented language to provide a general-purpose, dynamically extensible conferencing environment. However, LambdaMOO is totally centralised, and there are fundamental security problems in extending LambdaMOO (in its present form) to a distributed implementation. We are confident that Phantom will make it possible to implement a distributed conferencing system with all the flexibility and power of LambdaMOO, but with support for a graphical user interface, strong network security guarantees, and a distribution model which scales to support more users.

6 Acknowledgements

Special thanks to David Abrahamson, Luca Cardelli, Dan Connolly, Bill Janssen, Danny Keogan, Ciaran McHale, Killian Murphy and Brendan Tangney, who read early drafts of the language report and this paper, and provided valuable feedback on the exposition and language design.

References

[BL94] Tim Berners-Lee. Internet RFC 1738: Uniform Resource Locators (URL). URL: http://www.cis.ohio-state.edu/htbin/rfc/rfc1738.html, December 1994.

[BLC93] Tim Berners-Lee and Daniel W. Connolly. Hypertext Markup Language: A Representation of Textual Information and Metainformation for Retrieval and Interchange. URL: http://info.cern.ch/hypertext/WWW/MarkUp/HTML.html, 1993.

[BLCGP92] Tim Berners-Lee, R. Cailliau, J-F Groff, and B. Pollermann. World-Wide Web: The Information Universe. *Electronic Networking: Research, Applications and Policy*, 2(1):52–58, Spring 1992.

[BST89] Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3), September 1989.

[Car95] Luca Cardelli. A Language with Distributed Scope. In *Principles of Programming Languages*, January 1995. URL: http://www.research.digital.com/SRC/Obliq/Obliq.html.

[Cur92] Pavel Curtis. LambdaMOO Programmer's Manual. Technical report, Xerox Corporation, Palo Alto Research Centre, October 1992. URL: ftp://parcftp.xerox.com/ProgrammersManual.ps.

[Fab74] Robert S. Fabry. Capability-Based Addressing. *Communications of the ACM*, 17:403–412, July 1974.

[Gro92] The Object Management Group. The Common Object Request Broker: Architecture and Specification, 1992.

[IEE91] IEEE. *Std. 1178-1990. IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, 1991.

[IEE92] IEEE. *P1003.4a/D6 Threads Extension for Portable Operating Systems (Draft 6)*. Institute of Electrical and Electronic Engineers, February 1992.

[JSS94] Bill Janssen, Mike Spreitzer, and Denis Severson. Inter-Language Unification, 1.6.4. Technical Report P94-00058, Xerox Corporation, Palo Alto Research Centre, May 1994. URL: http://ftp.parc.xerox.com/pub/ilu/ilu.html.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, 1991.

[Ous94] John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[RJN94] David W. Robertson, William Johnston, and Wing Nip. Virtual Frog Dissection: Interactive 3D Graphics via the Web. In *Proceedings of the Second International WWW Conference*, 1994. URL: http://george.lbl.gov/ITG.hm.pg.docs/dissect/info.html.

[Ros92] Guido Van Rossum. Python Language Reference Manual. software documentation, 1992.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[SG90] James W. Stamos and David K. Gifford. Implementing Remote Evaluation. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[Str92] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1992.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. *ACM Sigplan Notices*, 2(12), 1987.

[WABL93] Edward Wobber, Martin Abadi, Mike Burrows, and Butler Lampson. Authentication in the Taos Operating System. Technical Report 117, Digital Equipment Corportation, Systems Research Centre, Decemeber 1993.