Service Combinators for Farming Virtual Machines

Karthikeyan Bhargavan\textsuperscript{1}, Andrew D. Gordon\textsuperscript{1}, and Iman Narasamdy\textsuperscript{2}

\textsuperscript{1} Microsoft Research
\textsuperscript{2} University of Manchester

Abstract. Management is one of the main expenses of running the server farms that implement enterprise services, and operator errors can be costly. Our goal is to develop type-safe programming mechanisms for combining and managing enterprise services, and we achieve this goal in the particular setting of farms of virtual machines. We assume each server is service-oriented, in the sense that the services it provides, and the external services it depends upon, are explicitly described in metadata. We describe the design, implementation, and formal semantics of a library of combinators whose types record and respect server metadata. We describe a series of programming examples run on our implementation, based on existing server code for a typical web application.

1 Introduction

Farms and Roles. Abstractly, a server farm is a collection of servers that runs one or more (parallel) programs, such as hosting a website or running compute jobs. Servers in a farm may have both local and remote dependencies. They may receive requests from remote clients, such as a web browser. They may also send requests to remote servers, to perform a credit card transaction, for example.

We assume each server boots off a disk image, such as the contents of a local hard drive, or an image fetched over the network. Typically, each server plays a particular role, such as web server, mail server, application server, and so on. The functionality embodied in disk images is often referred to as business logic, as it codifies the steps needed to enact various business processes—how to auction a wardrobe, for example.

Managing Server Farms in Software. Conventionally, operations staff manage server farms using a mixture of command prompts, scripts, various graphical tools, and actual physical configuration. Management includes provision and interconnection of servers, as well as responding to events such as peaks and troughs in load, or failures of individual servers. Operator errors are a leading cause of service failures and there is a need for increased automation and static validation of operator actions [Oppenheimer et al., 2003; Nagaraja et al., 2004].

Technologies such as network booting [Intel, 1999] and virtualization [Meyer and Seawright, 1970] of commodity hardware [Barham et al., 2003; Wolf and Halter, 2005] allow hardware resources to be dynamically allocated to server roles. Moreover, to eliminate physical configuration completely, virtual machines can even be rented on demand over the web [Bavier et al., 2004; Hoykhet et al., 2004; Amal, 2006; Garfinkel, 2007]. These technologies reduce the need for human intervention and transform server farm
management into a programming problem. The programming problem is to write operations logic: programs that codify the management actions of operators—to provision and interconnect servers, for example.

**Service Orientation.** Let a procedure be some functionality exposed on a communication port via a protocol such as RPC-style request/response, and let a service be a set of procedures. Server roles are increasingly service-oriented, in the sense that each role is described as importing and exporting services. A role implements its exports, and has dependencies on its imports. These imports and exports have explicit types that describe message contents and message patterns.

For example, an enterprise order processing application (drawn from published server code [Pallmann, 2005]) has an order entry role implementing a service (its export) consisting of a single procedure SubmitOrder, and conforming to the IOrderEntry interface. We give this and related interfaces below.

```csharp
public interface IOrderEntry { string SubmitOrder(Order order); }  
public interface IPayment { string AuthorizePayment(Payment payment); }  
public interface IOrderProcessing { void SubmitOrder(Order order); }
```

The code for SubmitOrder needs to consult a remote site to make an authorization decision. Hence, the order entry role has a dependency on the IPayment interface (its import). After authorization, SubmitOrder fulfills the order by calling an order processing procedure in the IOrderProcessing interface (its second import).

Service-oriented designs often use SOAP [Box et al., 2000] messages for requests and responses, while service metadata, such as request and response types, is often described using the Web Services Description Language (WSDL) [Christensen et al., 2001]. SOAP and WSDL are platform-independent XML formats. There are many development tools and software platforms for producing service-oriented disk images, where the imports and exports are described with WSDL. In our example, the order processing code is in C#. It relies on .NET communication libraries and tools to exchange SOAP messages and to map between C# interfaces and WSDL metadata.

**Service Combinators for Farming Virtual Machines.** If the server farm is the computer, what is the program? Our proposal is that an application running on a server farm consists of (1) a set of pre-existing disk images, (2) a set of URIs for the services exported and imported by the program, and (3) a script for assembling the roles, interconnecting them (sometimes via intermediaries for tasks such as load balancing), and managing the resulting system. The disk images implement the business logic of the program, while the script implements the operations logic.

Application for a Server Farm = Disk Images + External URIs + Script

Our main goals are (1) to develop a typed combinator-based API for scripting operations logic, and (2) to develop a formal semantics to support reasoning about reachable configurations. We treat service-oriented disk images as components to be interconnected using standard networking protocols.

On the other hand, the tasks of writing business logic and of building disk images are outside our scope—there are many existing tools for these purposes.
In this paper, we describe the design and implementation of Baltic, a type-safe API for expressing operations logic. Our API consists of a set of combinators for starting and stopping server roles implemented as virtual machines (VMs), for importing and exporting SOAP web services described by WSDL metadata, and for managing the resulting system. Concretely, the combinators are functions in the F# dialect of ML [Syme et al., 2007]. These combinators allow an ML program to control a small-scale server farm, which consists of a set of SOAP intermediaries interconnecting a set of VMs managed by a Virtual Machine Monitor (VMM) on a single physical server. By intermediary, we mean a service situated on the physical server, that performs simple tasks such as forwarding SOAP messages between VMs and the external world, or acting as a load-balancer in front of multiple servers. Our particular VMM is Virtual Server [Armstrong, 2007], running on dual processor hardware suitable for test automation or (modest) production workloads.

The intended scope of this paper is relatively small farms of servers, such as those that could be supported by a small number of VMMs. Our implementation is a research prototype, but if engineered for production, we believe it would usefully support small websites or test environments. We have not attempted to design a comprehensive set of intermediaries, although we can easily add more. Further research on scalability would be needed for our approach to apply to large-scale server farms used for parallel data processing (see Dean and Ghemawat [2004], for example). Still, even if our practical implementation is on a small scale, we demonstrate for the first time scripts that both (1) manipulate VMs and interconnect them with standard TCP/IP protocols, and (2) have a formal semantics suitable for typechecking and static analysis.

Contributions. We propose a resource metadata format that describes the resources—disk images, and imported and exported services—available in a server farm. We advocate that operations logic for managing a server farm be scripted with respect to such metadata. The main technical contributions of this work are the following.

– The idea that disk images should be seen as functions, with type signatures generated from service-oriented metadata, such as WSDL.
– The design and implementation of a library of service combinators for compositional description of server farms and their operations logic.
– A formal semantics of these combinators by an encoding in a concurrent λ-calculus. Via a type preservation result for our λ-calculus, we obtain type soundness for programs running against our API.

The main benefits of our approach, compared to the alternative, low-level scripting languages [Wolf and Halter, 2005], are the following.

– Our method abstracts from networking details and automatically links together the procedures imported and exported by machines and intermediaries.
– Our method catches construction errors by typechecking, rather than at some time during execution. For example, if server metadata stipulates a dependency on a service type, such as IPayment, we never supply a procedure with another type.

Contents of the Paper. Section 2 describes our software architecture. Section 3 introduces service combinators by example. Section 4 reviews WSDL service descriptions,
and describes our resource metadata format. Section 5 describes the implementation. Section 6 describes the message safety property guaranteed by Baltic, and outlines the underlying theory. Section 7 discusses related work and Section 8 concludes.

For the sake of brevity, this paper omits most formal definitions; a technical report [Bhargavan et al., 2007] includes additional examples, all formal definitions and proofs.

2 Architecture

The figure below depicts the Baltic architecture for managing a single physical server. A remote client is a consumer of a service located at an address on the physical server, while remote service is a service called by computations running on the physical server. The Baltic server implements the services exported by the physical server, as well as SOAP intermediaries that are used to interconnect other services. The VMM also runs on the physical server, under control of the Baltic server. (Our VMM is Virtual Server; other VMMs implement management APIs similar to that of Virtual Server, and hence could also support the Baltic API.) Files used by the VMM, such as disk images, are held on disks mounted on the physical server. The Baltic server mediates all access between VMs and the external world, and exports functions for managing the VMM and the intermediaries.

The Baltic script is an executable, S.exe, compiled from an ML program; it manages the Baltic server (and hence the VMM) using remote procedure calls, and hence may run either on the physical server, or elsewhere. The Baltic script is linked agains the libraries B.dll and E.dll that implement the Baltic API described in the next section.

The VMM hosts a virtual network to which each VM is attached. The virtual network is isolated from the external network. Hence, VMs can use TCP/IP over the virtual network to call services on other VMs directly. VMs may also use TCP/IP to call
intermediaries hosted by the Baltic Server, but cannot directly call remote services. Remote clients can also call externally accessible intermediaries hosted by the Baltic server, but not those hosted in VMs. Intermediaries hosted by the Baltic server can call all three kinds of procedure: other intermediaries, remote services, or services exported by a VM. As the diagram illustrates, Baltic scripts create and interconnect VMs and intermediaries, but are not (generally) involved in the actual SOAP message flow.

3 Service Combinators by Example

We implement several variations of the enterprise order processing application introduced in Section 1. First, we describe the resources available to the application and the typed Baltic API. Then we present a series of examples that manage the application by creating a different configuration of VMs and SOAP intermediaries.

Generating a Typed Interface to Resources. The resources available to an application consist of the following: (1) disk images for each server role; (2) addresses of external procedures that the application can use; and (3) addresses of procedures published by the application. We propose a metadata format to describe these resources, with service types being expressed in WSDL. The format is described in more detail in Section 4. (To the best of our knowledge, there is no prior service-oriented metadata format for disk images.) We show below an excerpt from the metadata for our example application—our implementation uses an XML format, but for the sake of readability we use an equivalent ML syntax.

```ml
[VM {name = "OrderEntry"; disk = "OrderW2K3.vhd";
    inputs = [payment_ty; orderProc_ty]; outputs = [orderEntry_ty]}];
Import {name = "Payment1"; binding = payment_ty;
    url = "http://creditagency1.com/CA/service.svc"};
Export {name = "OrderEntry"; binding = orderEntry_ty;
    url = "http://localhost:8080/OE/service.svc"};
...
```

The first record specifies that the OrderEntry role is defined by the disk image in the file OrderW2K3.vhd. The role takes two services as input, described by payment_ty and orderProc_ty, which are the WSDL descriptions (or bindings) corresponding to the C# interfaces IPayment and IOrderProcessing, given in Section 1. The role exports a single service described by the binding orderEntry_ty (corresponding to IOrderEntry).

The second and third records describe services imported and exported by the application; the records include the actual URIs as well as their WSDL bindings. The full metadata for our example contains also records for the other roles and imported services.

The table below lists the Baltic API for this example application. The Baltic API consists of a basic interface, B.mli, which is general and fixed, plus an environment interface, Em.mli, which depends on the resources described by m. (The notation Em.mli denotes an interface that is a function of the metadata m.) The types in Em.mli are ML representations of the request and response types in the WSDL bindings in m. The functions provide typed access to the various resources. Given m we have a tool that
compiles \(m\) to the interface \(E_m.mli\), and also to a module \(E_m-c.mli\) implements the interface. For compilation to succeed, the metadata must be well-formed in the sense that it satisfies certain syntactic constraints.

The functions in the Baltic API manipulate procedures; a value of type \((\alpha, \beta)\ proc\) represents a SOAP procedure that takes a request of type \(\alpha\) and returns responses of type \(\beta\). Section 5 describes the implementation and informal semantics of the API in more detail. In the rest of this section, we illustrate its use by example.

---

### Example 1: Creating an Isolated VM Farm.

Our first example creates a simple instance of the enterprise order processing application. The three server roles are implemented by VMs that are isolated from the external environment, a configuration useful during development and testing.

The script below calls `createPaymentRole` and `createOrderProcessingRole` to boot VMs from the disk images of the payment and order processing roles. The disk image is stored as an ordinary file, and a VMM can boot a VM off such a file. These calls return the procedures `ePay` and `eProc` exported by these roles. Since these roles import no procedures, the corresponding functions take no parameters. Finally, the third line boots a VM for the order entry role, using `ePay` and `eProc` as inputs.

```plaintext
let (vm1,ePay) = createPaymentRole ()
let (vm2,eProc) = createOrderProcessingRole ()
let (vm3,eEntry) = createOrderEntryRole ePay eProc
```

A distinctive feature of our approach is that instead of presenting disk images as untyped files, we generate code, like `createOrderEntryRole`, that presents disk images as functions manipulating typed procedures. Hence, typechecking catches interconnection errors that would otherwise cause failures at run time, either during initial configuration or later during reconfigurations.
The following function, from B.mli, makes a call to a procedure. Given an \((\alpha, \beta)\) proc and a request of type \(\alpha\), it serializes the request into a SOAP message, sends it to the procedure, awaits and then deserializes the response, and returns the result.

```ocaml
val call : (\alpha, \beta) proc \to \alpha \to \beta
```

This function is useful for testing; for example, to test eEntry, we invoke it as follows:

```ocaml
let o : Order = makeOrder "Alice"
let result = call eEntry o
```

The call function allows the Baltic script to send and receive SOAP messages; it is the only exception to the general rule that the Baltic script only sends control messages.

**Example 2: Importing and Exporting Services.** In our next example, rather than use a local payment service (such as ePay above) to authorize orders, we rely on a remote service. In addition, we export the internal eEntry procedure as a public service on the Baltic server. The external addresses of the payment service and the exported service are specified in the metadata \(m\), and are named Payment1 and OrderEntry. These addresses are embedded within the functions importPayment1 and exportOrderEntry in Em.mli.

The script below calls the function importPayment1 to create a SOAP forwarding intermediary (or forwarder) on the Baltic server, returning the internal procedure ePay. Any request sent to ePay is forwarded to the external URI Payment1. The call to the function exportOrderEntry with parameter eEntry creates another forwarder on the Baltic server, listening at the URI OrderEntry located on the physical server. Any request sent to this URI is forwarded to the internal procedure eEntry.

```ocaml
let ePay = importPayment1 ()
let (vm1, eProc) = createOrderProcessingRole ()
let (vm2, eEntry) = createOrderEntryRole ePay eProc
let eo = exportOrderEntry eEntry
```

**Example 3: Load Balancing between Server Instances.** If a server role becomes a bottleneck, we can avoid overloading it by running multiple instances in parallel, together with an intermediary that balances requests between them. The next example runs two instances of the front-end order entry role, to better utilize the multi-processor hardware on our server.

The script below calls createOrderEntryRole twice to create two VMs \(vm2\) and \(vm2'\) that export the procedures eEntry and eEntry', respectively. The function call eOr eEntry eEntry' then returns a procedure exported by a freshly created Or intermediary, which acts as a load balancer. Any request sent to this intermediary is forwarded to either eEntry or eEntry', chosen according to some strategy. (For now, we use a random strategy, but a more expressive API could allow multiple strategies.) The new procedure eor is then published at the external OrderEntry address as before.

```ocaml
let (vm2, eEntry) = createOrderEntryRole ePay eProc
let (vm2', eEntry') = createOrderEntryRole ePay eProc
let eor = eOr eEntry eEntry'
let eo = exportOrderEntry eor
```
The Or intermediary switches between two procedures. More generally, we can program derived combinators in ML; for example, orList creates a series of intermediaries to switch between a list of procedures by using the fold operator to compose a list of binary intermediaries.

\[
\text{let orList} : (\alpha, \beta) \text{ proc list} \rightarrow (\alpha, \beta) \text{ proc} = \text{List.fold1\_left eOr}
\]

Example 4: References, Updating References, and Events. In our previous examples, the communication topologies were fixed. Our next example introduces the idea of changing the topology in response to an event.

The combinator eRef e returns a procedure exported by a freshly created Ref intermediary, together with an identifier r for the intermediary. Any request sent to this intermediary is forwarded to e. Moreover, the intermediary r is mutable; a call to the combinator eRefUpdate r e’ updates r to forward subsequent requests to e’.

A VMM, such as Virtual Server, can detect various events during the execution of a VM, such as changes of VM state, the absence of a “heartbeat” (likely indicating a crash), and so on. Baltic provides a simple event handling mechanism, to allow a script to take action when an event is detected by the underlying VMM. The Baltic function eVM vm h associates the handler function h with the machine named vm. The handler function is of type event → unit where event is a datatype describing the event. (Our present implementation only handles a few kinds of events, but is extensible.)

To illustrate these operators, consider the two VM instances of the order entry role, vm2 and vm2’, in the previous example. If one of these VMs crashes, we should reconfigure our application to avoid sending messages to the crashed VM. The code in the following script creates a Ref intermediary that initially forwards messages to the eor procedure. If either VM crashes, an event handler updates the Ref intermediary to forward messages to the procedure exported by the other VM.

\[
\text{let (eref, r)} = \text{eRef eor}
\]

\[
\text{let h other ev = match ev with VM\_Crash \rightarrow eRefUpdate r other | _ \rightarrow ()}
\]

\[
\text{let _ = eVM vm2 (h eEntry’)}
\]

\[
\text{let _ = eVM vm2’ (h eEntry)}
\]

\[
\text{let eo = exportOrderEntry eref}
\]

Example 5: Snapshots of VMs. The current state of a running VM consists of its memory image plus the current contents of its virtual disk. Some VMMs, including Virtual Server, allow this state to be saved to disk; typically, the memory image is directly stored in one file, while the contents of the virtual disk are efficiently represented by a differencing disk, which records the blocks that have changed since the machine started. We refer to this file system representation of a VM state as a snapshot. A snapshot can be saved, perhaps multiple times, and subsequently restored.

Baltic includes a simple facility for saving and restoring snapshots. If vm is the name of a running VM, snapshotVM vm creates a snapshot, and returns a value ss of type vm\_snapshot that points to the saved files. Conversely, the call restoreVM ss discards the current state of vm, and replaces it by restoring the snapshot. (These combinators do not allow two snapshots of the same VM to run at once, a restriction imposed by the underlying VMM. However, the createRole functions in Em.mli can be called repeatedly to create multiple instances of any one role.)
As a variation of the previous example, we record a snapshot of vm2 and vm2’ just after booting and modify the event handler to restore the snapshot if the machine subsequently crashes. Snapshots allow faster recovery than rebooting.

```
let svm1 = snapshotVM vm2
let svm1’ = snapshotVM vm2’
let h ss ev = match ev with VM_Crash → restoreVM ss | _ → ()
let _ = eVM vm2 (h svm2)
let _ = eVM vm2’ (h svm2’)
```

The technical report [Bhargavan et al., 2007] has additional examples to illustrate how to program an array of replicated VMs, where the number of replicas varies depending on the load.

## 4 Metadata for Services and Resources

What we call **resource metadata** is a typed description of the disk images, imported services, and exported services of an application. We gave an example of metadata at the start of Section 3. The purpose of this section is to describe the general format. We begin with metadata for services, and use this to define metadata for resources.

### Service Metadata (WSDL)

We assume that imported and exported services are described in the WSDL metadata format [Christensen et al., 2001]. These WSDL files are generated automatically when the interface for the service is compiled, and are typically used to auto-generate proxy code for accessing the service.

A WSDL document describes a set of operations (procedures), and their input and output types. Types are typically expressed in XML Schema, though other formats are possible. We assume that the named types used within a WSDL document are captured as a set of abstract type declarations in an ML interface $I_y$, and that these abstract types have some concrete implementation $S_y$ corresponding to the XML Schema definition. There are several tools that map XML Schema descriptions to programming language types. In our example, $I_y$ consists of two abstract types $Payment$ and $Order$.

The following grammar is an abstraction of the WSDL syntax.

### WSDL Metadata for Services:

```
T_req, T_res type
n, a, d, u strings
O ::= {name = n, action = a, request_type = T_req, response_type = T_res}
Bd ::= {name = n, ops = $\{O_1; \ldots; O_m\}$} binding
P ::= {name = n, url = u, binding = Bd} port
```

We are using ML-style labelled records to represent the XML elements in a WSDL document. For example, our operations, bindings, and ports represent the WSDL elements named `<operation>`, `<binding>`, and `<port>`, respectively. For brevity, we sometimes elide the record labels when they are clear from context.
An operation \{n,a,T_{req},T_{res}\} describes a procedure, referred to as \(n\); a SOAP request to this procedure should have a header with SOAP action \(a\) and a body encoding a value of type \(T_{req}\), while a SOAP response from this procedure should have a body encoding a value of type \(T_{res}\).

A binding \{n,[O_1;\ldots;O_m]\} describes a service, referred to as \(n\), with \(m\) procedures described by \(O_1,\ldots,O_m\). For example, the payment ty binding used in our examples is defined as follows:

\begin{verbatim}
let payment ty: binding =
  {name = "Payment";
   ops = [{
         {name = "AuthorizePayment";
           action = "http://EOP/IPayment/AuthorizePayment";
           request_type = "Payment";
           response_type = "string";
        } : operation}]
}
\end{verbatim}

The payment ty binding describes a service, called Payment, that exposes a procedure AuthorizePayment, with a SOAP action http://EOP/IPayment/AuthorizePayment; the procedure takes as input an argument of type Payment and returns a string result.

A port \{n,u,Bd\} describes a service, referred to as \(n\); it is located at URI \(u\) and implements the procedures described in \(Bd\). For example, the following port describes an external service Payment1 that implements the payment ty binding and is located at the URI http://creditagency1.com/CA/service.svc.

\begin{verbatim}
let Payment1: port =
  {name = "Payment1";
   url = "http://creditagency1.com/CA/service.svc";
   binding = payment ty;
}
\end{verbatim}

Resource Metadata. Having defined the WSDL format for services, we define a metadata formal for a complete server farm.

### Metadata for Resources:

\[ r ::= \]

\[ r ::= VM\{n, disk = d, inputs = [Bd^{in}_1;\ldots;Bd^{in}_n], outputs = [Bd^{out}_1;\ldots;Bd^{out}_m]\} \]

\[ m ::= (I_{ty},S_{ty},[r_1;\ldots;r_n]) \]

Let \(rs\) be the record list at the start of Section 3; it is an example of a resource list.

Each VM record defines a role in terms of a VM name, a disk image file accessible from the VMM, a list of imported bindings, and a list of exported bindings. In our example, the OrderEntry, OrderProcessing, and Payment roles are defined by such records.

Each Import record defines an external service port that we wish to use from within the Baltic server. In our example, the Payment1 port is imported by such a record.

Each Export record defines an internal service port that we wish to make available externally. In our example, the OrderEntry port is published at the public URI http://localhost:8080/OE/service.svc by such a record.
Recall that $I_V$ consists of the abstract types Payment and Order in our example. Let $S_V$ be an implementation of these two abstract types, for example, a couple of record types. Then $m = (I_V, S_V, rs)$ is the metadata for our examples. In general, the ML interface $I_V$ and corresponding ML implementation module $S_V$ are present simply to define types used in the resource list $rs$.

5 Implementation and Informal Semantics of the Baltic API

An implementation of the Baltic API consists of a module $B-.c.ml$ implementing the basic interface $B.mli$ and a module $Em-.c.ml$ implementing the metadata-specific environment interface $Em.mli$. These modules are compiled to generate the libraries $B.dll$ and $Em.dll$ respectively; ML programs linked with these libraries are compiled to assemble Baltic scripts that manage the Baltic server.

In addition to this concre te implementation, we also describe a symbolic implementation of the Baltic API, consisting of the modules $B.ml$ and $Em.ml$. These symbolic modules simulate the behaviour of the Baltic server in terms of local processes and channel-based communications; as such, they constitute our executable semantics of the API. An ML program compiled with these modules generates a symbolic executable that can be used to trace and debug a Baltic script before deployment.

Basic Module: $B-.c.ml$. The module $B-.c.ml$ implements the basic interface $B.mli$ by managing intermediaries and VMs on the Baltic server. To manage SOAP intermediaries, we use the web services functionality provided by the Microsoft .NET Framework API; to manage VMs we rely on functions in the Virtual Server API. We outline our implementation of the types and functions in the $B.mli$ below. The technical report [Bhargavan et al., 2007] contains further details.

- A value of type $vm$ is a VM identifier, as defined by the VMM.
- A value of type $vm.snapshot$ is the name of a directory containing a group of files implementing a VM snapshot, together with a VM identifier.
- A value of type $event$ is one of four events detected by the VMM: either a VM has crashed, or shut down, or its processor is overloaded (running close to full capacity), or its processor is underloaded.
- A value of type $(\alpha, \beta)$ proc is a SOAP address, consisting of a URI and a SOAP action, and located either on a VM or the physical server. The API generates a value of this type only when there is a web service of the appropriate type listening at the address.
- A value of type $(\alpha, \beta)$ procref is a mutable variable on the physical server storing the SOAP address of a procedure of type $(\alpha, \beta)$ proc.
- The function call $call e a$ implements a remote procedure call: it takes a procedure $e$ of type $(\alpha, \beta)$ proc and an argument $a$ of type $\alpha$; it sends $a$ as a request to $e$ and returns the response.
- The function call $eOr e1 e2$ takes two procedures $e1$ and $e2$ and creates and returns a fresh address $e$ on the physical server; it starts an intermediary on the Baltic server that listens for requests on $e$ and forwards them either to $e1$ or to $e2$ (based on a coin-toss); the intermediary waits for the corresponding response and returns it.
The function call \textit{ePar} \textit{e1} \textit{e2} also takes two procedures \textit{e1} and \textit{e2} and creates and returns a fresh address \textit{e} on the physical server; it starts an intermediary on the Baltic server that listens for requests on \textit{e} and forwards them to both \textit{e1} and \textit{e2}; the intermediary waits for the first response and returns it.

The function call \textit{eRef} \textit{ei} takes a procedure \textit{ei} and creates and returns a fresh address \textit{e} on the physical server plus a new mutable variable \textit{r} on the Baltic server that stores the address of the procedure \textit{ei}; it starts an intermediary on the Baltic server that listens for requests on \textit{e} and forwards them to the address currently stored in \textit{r}.

The function call \textit{eRefUpdate} \textit{r} \textit{e} modifies the variable \textit{r} on the Baltic server to point to the procedure \textit{e}.

The function call \textit{eVM} \textit{vm} \textit{h} takes a VM identifier \textit{vm} and an event handler \textit{h}; it registers this handler at the Baltic server so that whenever the VMM detects an event \textit{ev} for the VM \textit{vm} the handler \textit{hev} is executed.

The function call \textit{shutdownVM} \textit{vm} shuts down the VM with identifier \textit{vm}.

The function call \textit{snapshotVM} \textit{vm} saves a snapshot of the current state of the running VM \textit{vm} in a new directory \textit{d} and returns a value of type \textit{vm\_snapshot} containing \textit{d} and \textit{vm}.

The function call \textit{restoreVM} \textit{ss} checks that \textit{ss} contains a directory \textit{d} containing a valid snapshot of VM \textit{vm}; it shuts down any running VM with identifier \textit{vm} and starts up a VM from the running state stored in \textit{d}.

**Environment Module: \textit{Em\_c.ml}**. The module \textit{Em\_c.ml} which enables access to the resources described in the metadata \textit{m}. The technical report [Bhargavan et al., 2007] describes a tool that compiles \textit{m} to the interface \textit{Em.mli} and to the module \textit{Em\_c.ml}. Given metadata \textit{m} = (\textit{Ity}, \textit{Sty}, \textit{rs}), \textit{Em\_c.ml} is implemented as follows:

- It contains the type definitions \textit{Sty}.
- For every VM record in \textit{rs} with name \textit{N}, disk image \textit{d}, inputs of type \textit{s1},...,#\textit{sn} and outputs of type \textit{t1},...,#\textit{tm}, the function call \textit{createNRole} \textit{i1}...\textit{in} takes \textit{n} services \textit{i1}...\textit{in} (of type \textit{s1},...,#\textit{sn}) as arguments, boots a new VM \textit{vm} from the disk image \textit{d}, configures \textit{vm} with the SOAP addresses of its inputs \textit{i1}...\textit{in}, and returns \textit{vm} plus its exported services \textit{o1}...\textit{om} (of type \textit{t1},...,#\textit{tm}). For example, the function call \textit{createOrderEntryRole} \textit{ePay} \textit{eProc} takes two procedures \textit{ePay} and \textit{eProc} as arguments; it then boots a new VM \textit{vm} from the disk image file "OrderW2K3.vhd", configures \textit{vm} with the addresses of \textit{lsPay} and \textit{eProc}, and returns \textit{vm} and the address of the new order entry procedure exported by it.
- For every Import record in \textit{rs} with name \textit{N} and url \textit{U}, the function call \textit{importN} () creates and returns a fresh address \textit{e} on the physical server; it starts an intermediary on the Baltic server that listens for requests on \textit{e} and forwards them to the external url \textit{U}, waits for the corresponding response, and returns it.
- For every Export record in \textit{rs} with name \textit{N} and url \textit{U}, where \textit{U} is an externally accessible address on the physical server, the function call \textit{exportN} \textit{e} starts an intermediary on the Baltic server that listens for requests at \textit{U} and forwards them to the procedure \textit{e}, waits for the corresponding response, and returns it.

**Symbolic Modules: \textit{B.ml} and \textit{Em.ml}**. The modules \textit{B.ml} and \textit{Em.ml} simulate the behaviour of the implementaion modules \textit{B\_c.ml} and \textit{Em\_c.ml}, but without contacting
the Baltic server or the VMM and without sending any messages on the network. Instead, they model VMs and intermediaries as local processes spawned by the script, and implement SOAP requests and responses as local channel-based communications between processes.

A value of type \((\alpha, \beta)\) proc is modelled as a function that takes values of type \(\alpha\) and returns values of type \(\beta\); a value of type \((\alpha, \beta)\) procref is a mutable reference to such a function. Hence, in B.ml, the function call eOr e1 e2 creates and returns a new function \(f\); when \(f\) is called with an argument \(v\), it calls either \(e1\) or \(e2\) with \(v\) (based on a coin-toss) and returns the result. The functions call, ePar, eRef, and eRefUpdate are implemented similarly.

A VM is modelled as a partition: a named collection of processes sharing state in the form of local communication channels. A value of type \(vm\) is a name of a partition plus a channel on which events for the VM are triggered. Hence, in Em.ml, the function call createNRole \(i_1...i_n\) creates a new VM consisting of a partition \(a\) and a fresh channel \(ev\); \(a\) contains newly spawned processes that use the procedures in \(i_1...i_n\) to implement the exported procedures \(o_1...o_m\); the processes in \(a\) may also send events on \(ev\). For example, the function call createOrderEntryRole \(ePay\) \(eProc\) creates a partition \(a\) with a single process that listens for order requests on a fresh local channel \(c\), then calls the payment procedure \(ePay\) and the order processing procedure \(eProc\) before returning a response; createOrderEntryRole returns the partition name \(a\) and a fresh event channel \(ev\); it also returns an output procedure \(f\) that when given an argument \(v\), sends \(v\) on the channel \(c\) within \(a\) and returns the response. The function call eVM \(vm\) \(h\) spawns a process that listens for events \(e\) on the event channel corresponding to \(vm\) and executes \(he\). A VM snapshot contains the state of a partition; hence snapshotVM \(vm\) saves the current values of all the channels and processes of the partition corresponding to \(vm\); restoreVM \(ss\) restores a saved partition.

Each imported service \(N\) in \(m\) is modelled as a process listening on a global channel \(Nchan\). For example, the external payment service \(Payment1\) is modelled as a process listening for requests on the channel \(Payment1Chan\). In Em.ml, the function call import\(N()\) returns a procedure that takes an argument \(v\), forwards it on to \(Nchan\), and returns the response. Conversely, for each exported service \(N\) in \(m\), the function call export\(N\) \(e\) listens for requests on a global channel \(Nchan\) and forwards them to the procedure \(e\).

6 Message Safety

Since the APIs available to a Baltic script are strongly typed, any system of VMs and intermediaries assembled by a Baltic script is well-typed by construction. In this section, we give an informal description of message safety and its proof by typing. The formal details are in a technical report [Bhargavan et al., 2007].

Let an entity be any source of a SOAP message; entities include remote clients and servers, intermediaries on the Baltic server, VMs, and the Baltic script itself.

Let an entity respect a procedure of type \((\alpha, \beta)\) proc if and only if

1. each SOAP message sent by the entity to the procedure has type \(\alpha\); and
2. each SOAP message sent by the entity in response to a message to the procedure has type \(\beta\).
The desired property of a system generated by a Baltic script is the following.

**Message Safety Property.** For any metadata \( m \), if

1. the Baltic script is well-typed against \( E.m\text{.mli} \) and \( B.m\text{.mli} \)
2. all remote entities respect the procedures in \( m \), and
3. disk images respect the procedures they import and export
then all entities arising during a run respect all procedures.

Many interconnection errors, where servers or intermediaries are connected to the wrong addresses, lead to entities not respecting procedures. Our safety property guarantees, by static type-checking, that such errors cannot arise.

Assuming points (1), (2), and (3) of the Message Safety Property, we can construct a well-typed ML expression that represents the message-passing behaviour of a complete Baltic system. To do so, we generate an ML interface \( X.m\text{.mli} \) consisting of typed ML channels to represent each of the procedures exported and imported by \( m \). In the case of our running example, this interface is as follows:

```ml
val Payment1: (Payment × string) chan
val Payment2: (Payment × string) chan
val OrderEntry: (Order × string) chan
```

Our ML expression for the whole system is the composition \( B.m\text{.ml} E.m\text{.ml} S.m\text{.ml} O.m\text{.ml} \) of the following modules.

1. \( B.m\text{.ml} \) is the symbolic implementation of the \( B.m\text{.mli} \) interface. It has no dependencies.
2. \( E.m\text{.ml} \) is the symbolic implementation of the \( E.m\text{.mli} \) and \( X.m\text{.mli} \) interfaces. It depends on the interface \( B.m\text{.mli} \) provided by \( B.m\text{.ml} \).
3. \( S.m\text{.ml} \) is the Baltic script. It depends on the interfaces \( B.m\text{.mli} \) and \( E.m\text{.mli} \) provided by \( B.m\text{.ml} \) and \( E.m\text{.ml} \).
4. \( O.m\text{.ml} \) represents the remote entities, that is, the external clients and services. It depends on the interface \( X.m\text{.mli} \) provided by \( E.m\text{.ml} \).

Since the dependencies of each of the modules in \( B.m\text{.ml} E.m\text{.ml} S.m\text{.ml} O.m\text{.ml} \) are provided by preceding members, the whole composition is well-typed. The message safety property can then be obtained from type safety for ML.

We formalize this argument in the technical report [Bhargavan et al., 2007]. In fact, to model VMs with snapshots we need to develop a concurrent \( \lambda \)-calculus, called the *partitioned \( \lambda \)-calculus*. We prove type safety theorems for the *partitioned \( \lambda \)-calculus*. By appeal to these theorems, we formalize the composition \( B.m\text{.ml} E.m\text{.ml} S.m\text{.ml} O.m\text{.ml} \) as a \( \lambda \)-calculus expression, and hence prove the Message Safety Property as a theorem about the partitioned \( \lambda \)-calculus.

### 7 Related Work

**Related Systems.** Edinburgh LCFG [Anderson, 1994] is a well-developed system for managing the configuration of large numbers of Unix-like machines. LCFG can configure software within disk images, a task not addressed by the Baltic operators. On the
other hand, LCFG does not support intermediaries, and uses an untyped scripting language, while Baltic introduces the idea of representing server roles as typed functions.

HP SmartFrog [Goldsack et al., 2003] is a domain-specific language for describing server components, together with an implementation for activating and managing them. The original version worked with JVM-based components, but a more recent version uses operating system virtualization. Like LCFG, SmartFrog can describe the structure of server roles. SmartFrog has a type system, but is not service-oriented, in the sense of treating a server role as importing and exporting typed procedures.

The AppLogic grid operating system [3TERA 2006] allows VM server farms to be constructed and managed with a graphical editor. AppLogic grids are configurable using conventional scripting languages.

HPorter [Huang et al., 2007] is another combinator library written in a functional language for combining and reconfiguring software components written in lower-level languages. HPorter is aimed at robotics applications, and manages pre-existing components written in C and C++, that communicate over TCP/IP sockets. HPorter is written in the pure functional language Haskell, and relies on Haskell’s higher-order type theory to encapsulate imperative behaviour.

Like Baltic, PiDuce [Carpineti et al., 2006] is a language and implementation for building SOAP web services, with a formal semantics. Unlike Baltic, PiDuce expresses the behaviour of individual services directly, whereas Baltic relies on pre-existing disk images to implement individual services, and concentrates on management.

Related Formalisms. We build both our actual implementation and our formal semantics using the technique of dual concrete and abstract implementations of interfaces; this technique was introduced by Bhargavan et al. [2006].

Our use of the \( \lambda \)-calculus with partitions as a semantics for combinations of virtual machines is a refinement of an earlier proposal, by Gordon [2005], that operating system virtualization can usefully be formalized using process calculi. There are other process calculi with operators to snapshot, restore, and duplicate running locations, including the Kell Calculus [Schmitt and Stefani, 2003; Lienhardt et al. 2007] and the Seal Calculus [Castagna et al., 2005]. A great many formalisms—see Lapadula et al. [2007], for example, and its bibliography—have been developed to represent orchestration, choreography, and dynamic discovery of web services. We do not address these advanced topics, and instead focus on management of pre-existing systems using simple request/response patterns of SOAP messaging; such systems are the common case in server farms today.

8 Conclusions and Future Work

We have described a set of combinators for assembling networks of virtual machines that export SOAP services. A combination of typechecking together with automated allocation of addresses prevents the troublesome configuration errors that may arise with alternatives, such as untyped scripts. There is a semantics based on a typed concurrent \( \lambda \)-calculus with partitions, and an implementation using Virtual Server with scripts in ML. Our test scripts manage pre-existing components from a sample multi-tier web application.
In future work, we intend to address some of the limitations in our present implementation. Our implementation does not consider security (we are essentially trusting the code on disk images), or the control of multiple VMMs (for performance and fault tolerance), or persistent state (any transient changes to disk images are discarded). A lightweight mechanism to customize each instance of the same disk image would be useful. Intermediaries are limited to SOAP request/responses and do not maintain SOAP-level sessions. We support SOAP services but neither arbitrary webpages nor database connections. On the basis of our formal semantics, we intend to develop techniques for reasoning about operations logic expressed using our combinators.

Acknowledgments. Conversations with Úlfar Erlingsson, Philippa Gardner, Galen Hunt, Dave Langworthy, Alan Schmitt, Clemens Syzperski, and Paul Watson were useful. Questions raised during a presentation of a preliminary version of Baltic at the Semantics Lunch at the University of Cambridge Computer Laboratory were extremely useful; thanks are due in particular to Andy Pitts, Peter Sewell, and Viktor Vafeiadis. Galen Hunt suggested the term “operations logic”.

Bibliography

AppLogic: Grid Operating System for Utility Computing. 3TERA (September 2006),
http://0301.netclime.net/1_5/8/A/8/3teraAppLogic0906.pdf
Amazon Elastic Compute Cloud (Amazon EC2) - Limited Beta. Amazon Web Services LLC
(August 2006), http://aws.amazon.com/ec2
8th Large Installations Systems Administration (LISA) Conference, Berkeley, CA, pp. 19–26
(1994)
Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I.,
Warfield, A.: Xen and the art of virtualization. In: Symposium on Operating Systems
Bavier, A., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., Wawr-
zoniak, M.: Operating system support for planetary-scale network services. In: NSDI 2004
(2004)
Bhargavan, K., Fournet, C., Gordon, A.D., Tse, S.: Verified interoperable implementations of
Bhargavan, K., Gordon, A.D., Narasamdy, I.: Service combinators for farming virtual machines.
Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H., Thatte, S.,
Carpineti, S., Laneve, C., Padovani, L.: Piduce—a project for experimenting web services tech-
nologies (2006), http://www.cs.unibo.it/PiDuce/
Castagna, G., Vittek, J., Zappa Nardelli, F.: The seal calculus. Information and Computa-
tion 201(1), 1–54 (2005)
Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: Web Services Description Language
(WSDL) 1.1 (2001)
Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Sixth Sym-
Garfinkel, S.: Commodity grid computing with Amazon’s S3 and EC2. login, 7–13 (February 2007)


