Baltic: Service Combinators for Farming Virtual Machines

Andy Gordon, Microsoft Research

Joint work with
Karthik Bhargavan, Microsoft Research
Iman Narasamdya, University of Manchester

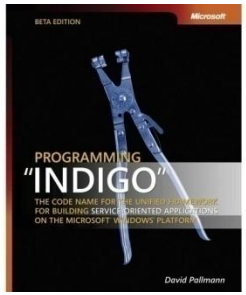
Fun in the Afternoon, May 2007

Farms, Roles, Service Orientation

- Abstractly, a **server farm** is a set of servers.
- Each server boots off a **disk image**, eg, the contents of a local hard drive, or an image fetched off the network.
- Each server plays a particular **role**, eg, web server, database.
- We assume the disk images, and hence the behaviour, of each server in a particular role are essentially the same.
- A **service** is a set of **endpoints**, each of which is a communication port implementing a function accessible by RPC
- Server roles are increasingly **service-oriented** in that they are described as **importing** and **exporting** typed endpoints.
- For example, Web Services Description Language (WSDL) is a common format for describing services based on typed SOAP messages.

Three Example Roles

- Our scripting examples concern three web services used as the back end of a web app, the "Adventure Works" website
- The code, from a book on programming .NET 3, is in C#
- Each of the services consists of a single endpoint
- These services have standard WSDL metadata



OrderEntry	Payment	OrderProcessing
• called when a customer completes checkout • OrderW2K3.vhd	• called to authorize payment • PaymentW2K3.vhd	• called to fulfill the order • ProcW2K3.vhd

WSDL Metadata for Services

```
let payment:service =
  {sname = "Payment";
   ops = [{opname = "AuthorizePayment";
           action = "http://tempuri.org/IPayment/AuthorizePayment";
           input = "ProgrammingIndigo.Payment";
           output = "string"}]}
let orderProc:service =
  {sname = "OrderProcessing";
   ops = [{opname = "SubmitOrder";
           action = "http://AdventureWorks/IOrderProcessing/SubmitOrder";
           input = "ProgrammingIndigo.Order";
           output = "unit"}]}
let orderEntry:service =
  {sname = "OrderEntry";
   ops = [{opname = "SubmitOrder";
           action = "http://AdventureWorks/IOrderEntry/SubmitOrder";
           input = "ProgrammingIndigo.Order";
           output = "string"}]}
```

Operating System Virtualization

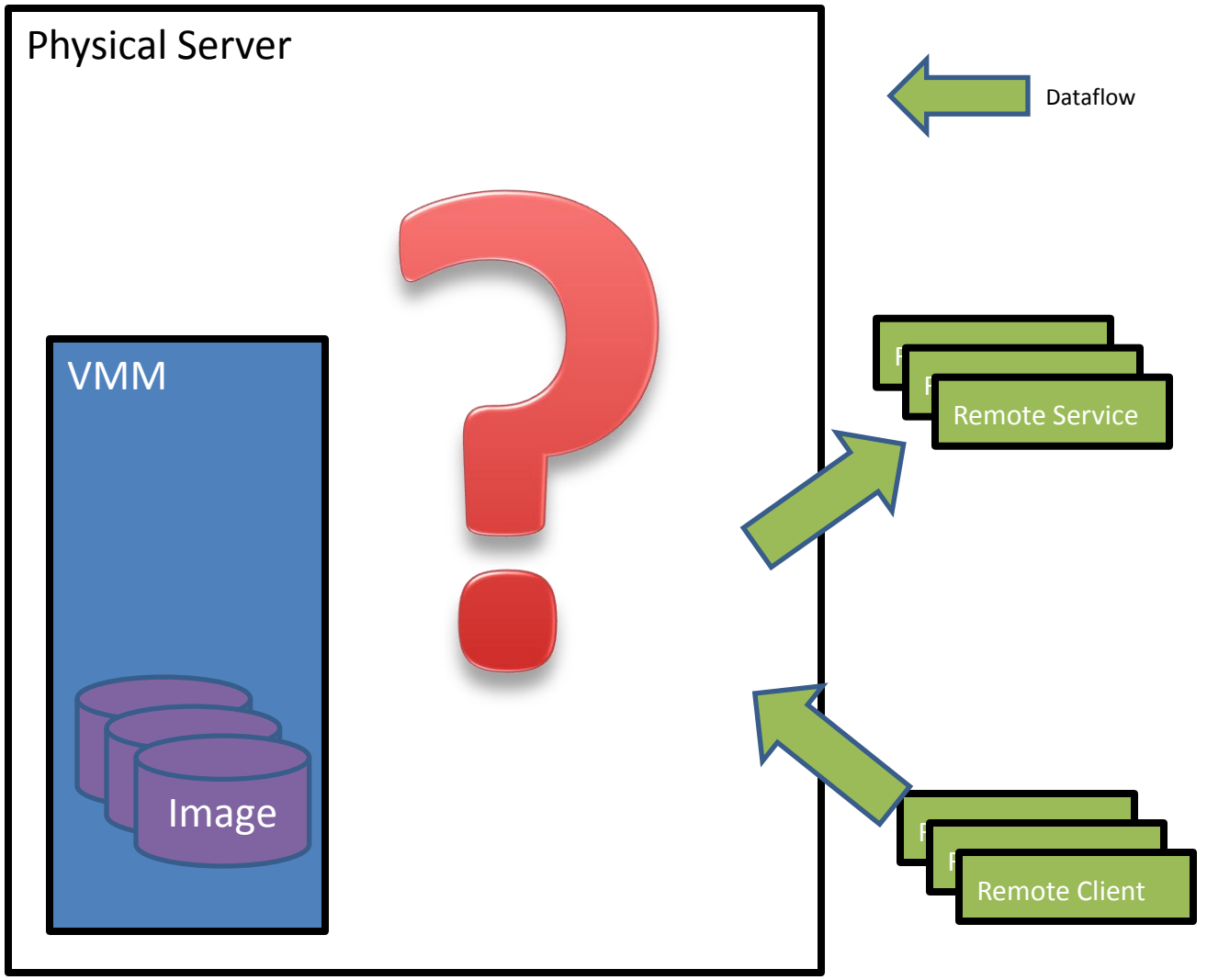
- A **host** OS runs on physical machine and controls its devices, and runs multiple **guest** OSs within VMs
  - Control software known as **virtual machine monitor** (VMM)
  - Guests' disk images held as **virtual hard disk** files on the host
  - VMs attached to **virtual networks**, which may or may not be isolated
- VMM gives guests the illusion of direct access to hardware
  - Not object-oriented runtime with garbage collection, as provided by a language-based VM such as the JVM or CLR
- VMware launch first VMM on commodity x86 hardware in 1999; first server product in 2001
  - Other x86 VMMs include Xen, KVM, and Microsoft Virtual Server
  - OS virtualization on mainframes goes back to VM/370, launched 1972

The Programming Problem

Goal: export services on physical server to remote clients

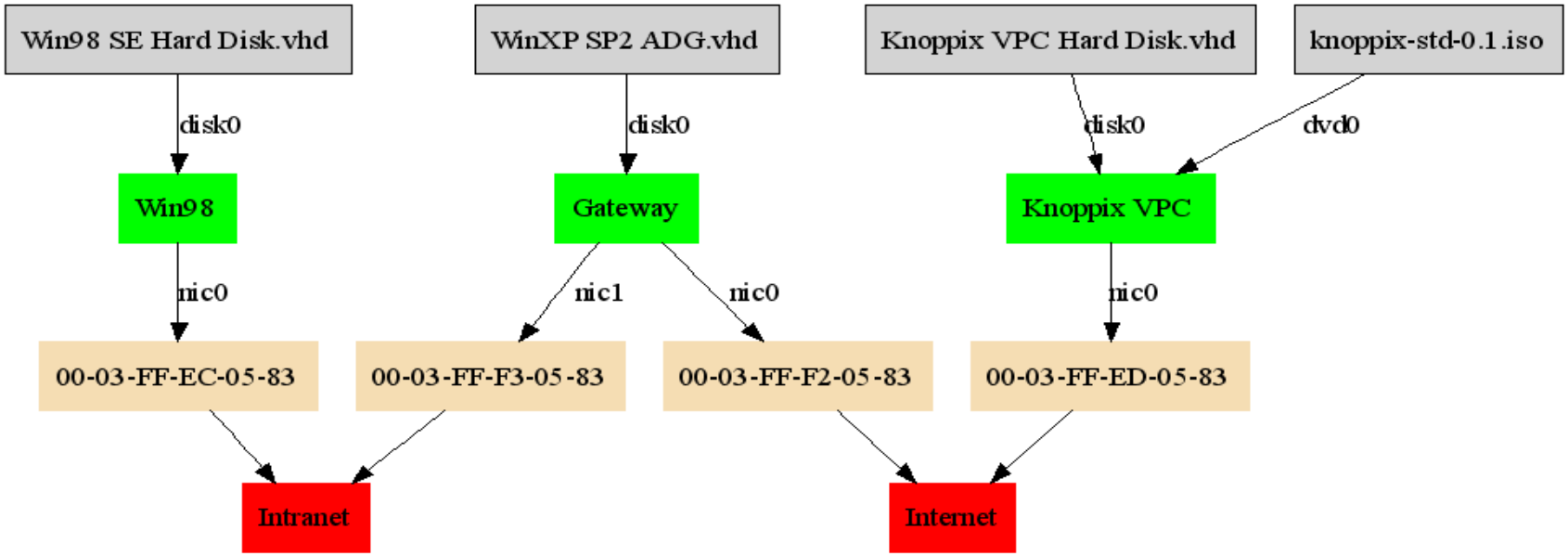
Resources: pre-compiled disk images and remote services

Problem: how to program a VMM to achieve the goal with the resources



Existing APIs: Device Combinators

- Existing APIs for VMMs manipulate a **wiring graph**
  - Each node is a device, eg, disk, machine, network adapter, network, etc
  - Each edge is virtual wiring between devices



Metadata for Disks and I/O

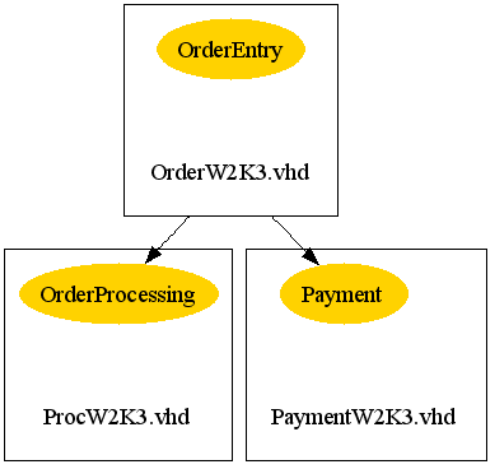
```
let m:metadata =
  [VM {vmname = "OrderEntry"; disk = "OrderW2K3.vhd";
       inputs = [payment; orderProc]; outputs = [("/OrderEntry.svc",orderEntry)]};
  ...
  Import {name="Payment1“; url = http://credit.fr/CA/service.svc ;  service=p ayment};
  ...
  Export {name="OrderEntry“;url = http://localhost:80/OE/service.svc ; service=orderEntry}]
```

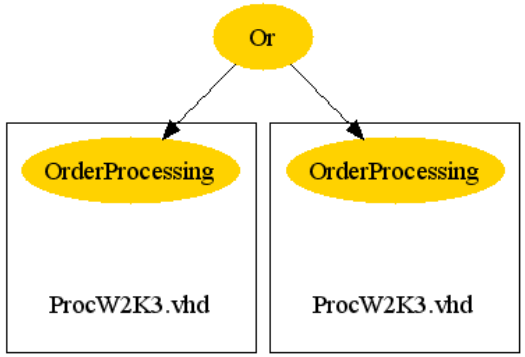
Our Observation: our resources – disks, imports, and exports – can be assigned service-oriented metadata

Our Hypothesis: typed interfaces to our resources, generated from this metadata, are more productive and less error-prone than conventional device-oriented APIs

Baltic API: Service Combinators

- Our API for VMMs manipulates a **call graph**
  - Each node is an endpoint, ie, an RPC implementation, hosted either within a VM or as a separate intermediary
  - Each edge represents a potential RPC
- Each VM is shown as a box
  - the endpoints in the box are its **exported service**
  - the endpoints connected to the box are its **imported service**



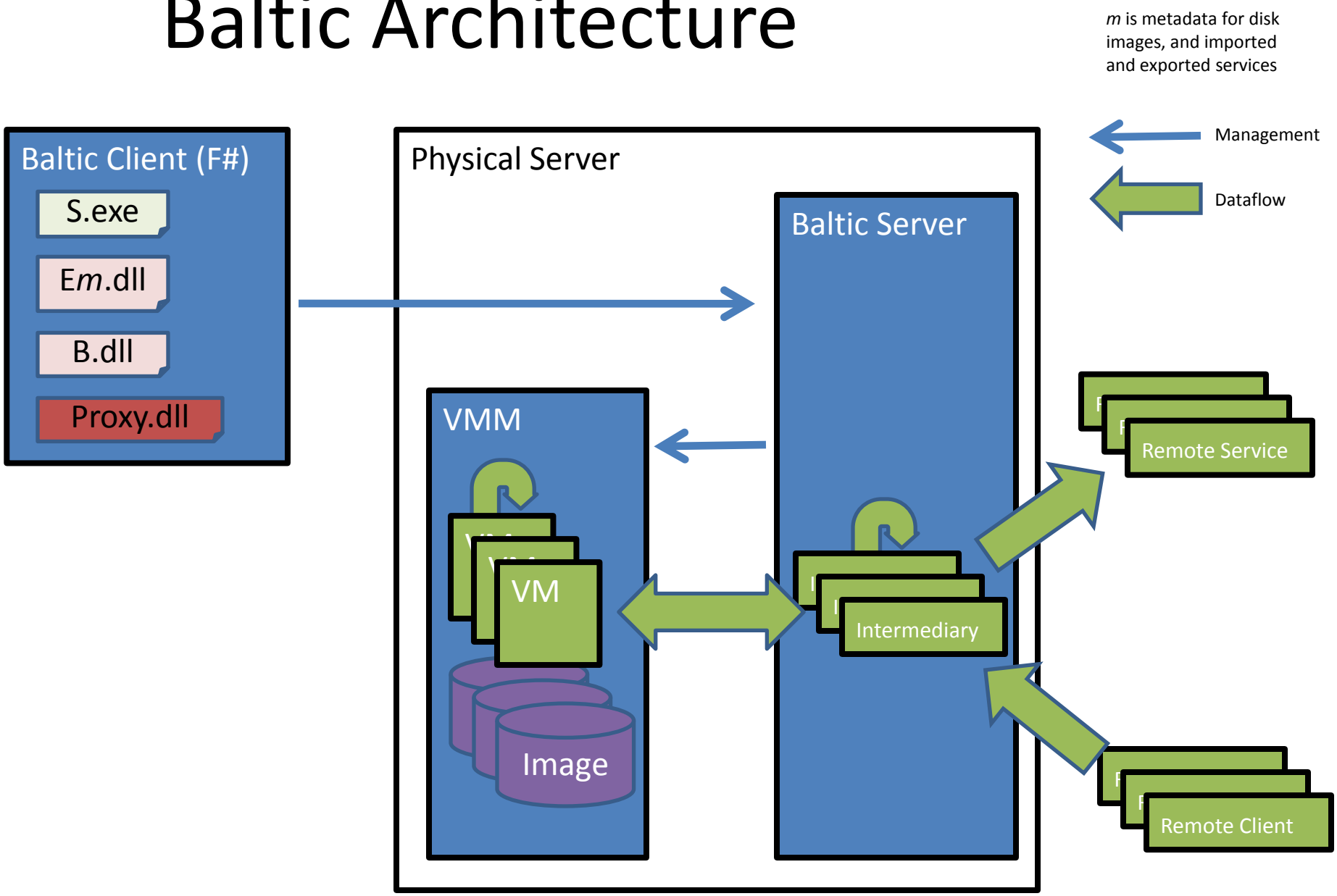


Baltic: Service Combinators

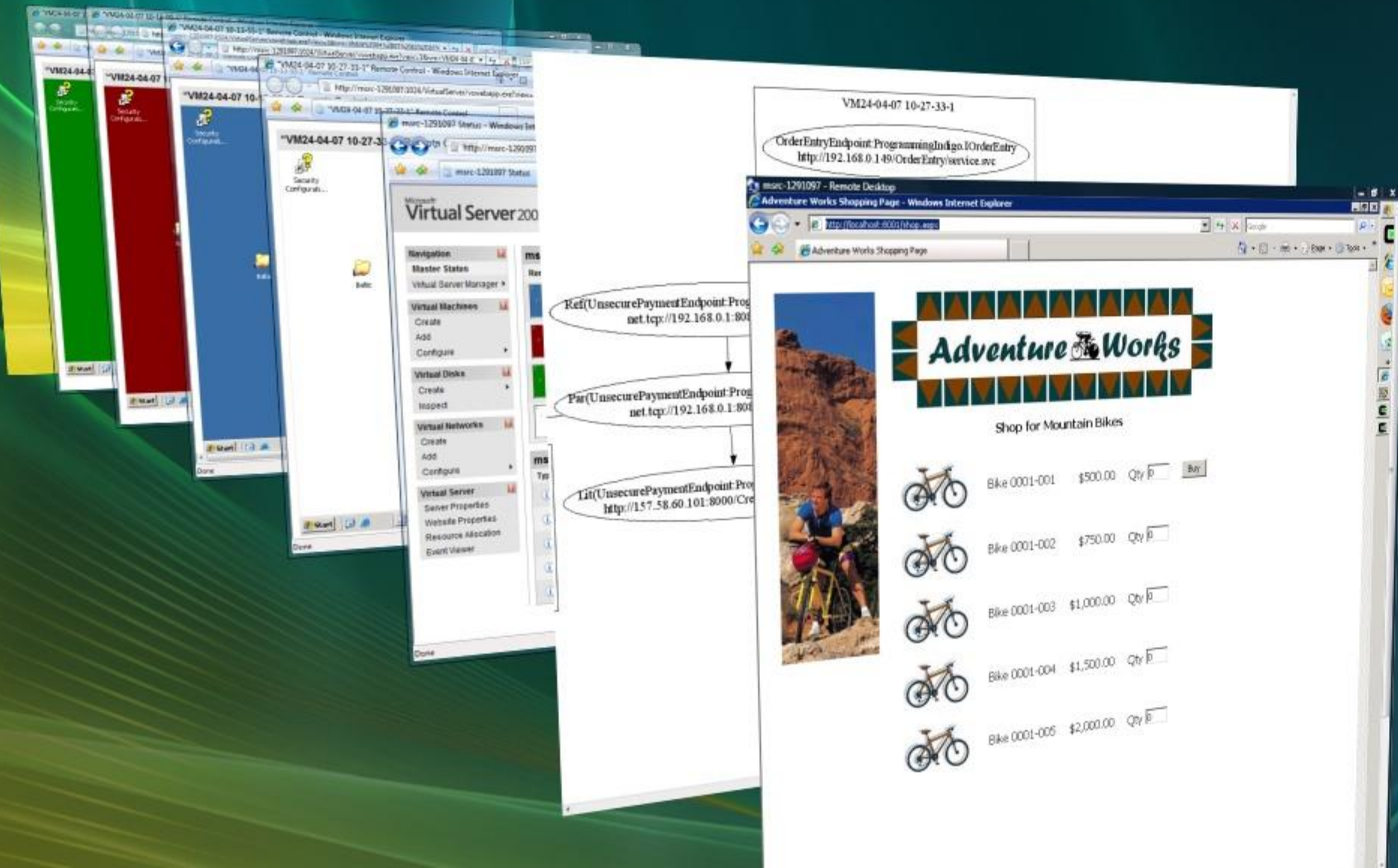
- VMM APIs are an essential piece of the farm management puzzle
- Current **device-oriented APIs** are undoubtedly usable, but:
  - They are low-level – virtual devices and virtual wiring
  - They ignore service-oriented metadata – VMs are simply nodes
  - They are error-prone – no protection from connection errors
- Instead, we propose a higher-level **service-oriented API**
  - Our combinators operate on a call graph – endpoints and potential calls – but are implemented over a device-oriented API
  - We exploit service-oriented metadata (eg WSDL) to generate typed descriptions of server roles and external endpoints – VMs are functions from imported service to exported service
  - A benefit of typing is that various connection errors are detected statically, rather than sometime during execution.

BALTIC ARCHITECTURE, COMBINATOR EXAMPLES

Baltic Architecture



4 VMs, VMM, Baltic, External Client



B.mli

```
type vm
type vm_snapshot
type event = VM_Crash
type ('a,'b) endpoint
type ('a,'b) endpointref
val eOr : ('a,'b) endpoint -> ('a,'b) endpoint -> ('a,'b) endpoint
val ePar : ('a,'b) endpoint -> ('a,'b) endpoint -> ('a,'b) endpoint
val eRef : ('a,'b) endpoint -> ('a,'b) endpoint * ('a,'b) endpointref
val eRefUpdate : ('a,'b) endpointref -> ('a,'b) endpoint -> unit
val eVM : vm -> (event -> unit) -> unit
val snapshotVM : vm -> vm_snapshot
val restoreVM : vm_snapshot -> unit
```

Em.mli

```
type tPayment = (Payment,string) endpoint
type tOrderEntry = (Order,string) endpoint
type tOrderProcessing = (Order,unit) endpoint
val createOrderEntryRole : tPayment -> tOrderProcessing -> (vm * tOrderEntry)
val createOrderProcessingRole : unit -> (vm * tOrderProcessing)
val createPaymentRole : unit -> (vm * tPayment)
val importPayment1 : unit -> tPayment
val importPayment2 : unit -> tPayment
val exportOrderEntry : tOrderEntry -> unit
```

Endpoints and Services

```
type Payment = { ... }
type Order = { ... }

type tOrderEntry = (Order,string) endpoint          // OrderEntry service
type tOrderProcessing = (Order,unit) endpoint       // OrderProcessing service
type tPayment = (Payment,string) endpoint           // Payment service
```

- A value of type (α,β) *endpoint* is the network address of a SOAP RPC server, reachable from VMs and the physical host
- (Only endpoints on the physical host are externally reachable)
- A service is (in general) a tuple of endpoints.
- The type of each service is an instance of the scheme:

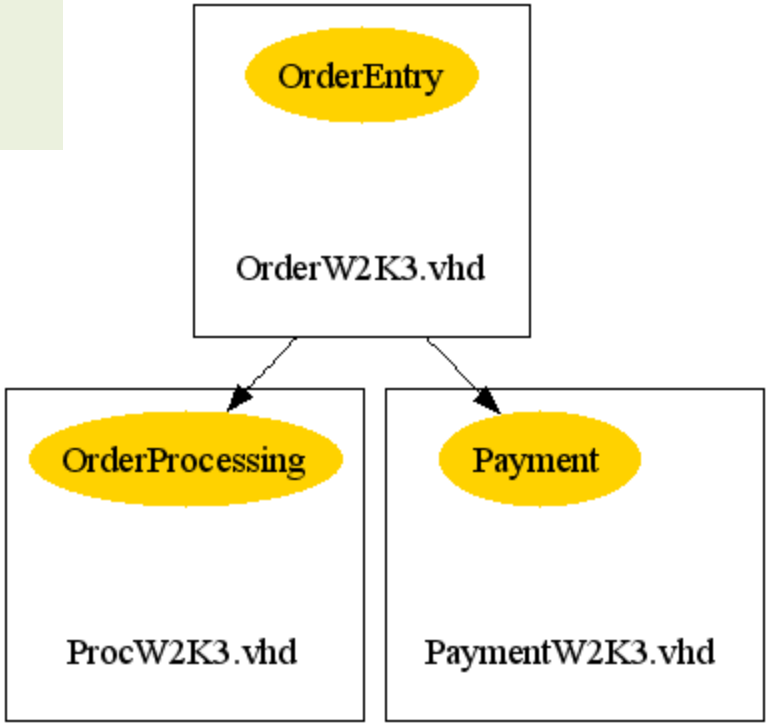
  (α_1,β_1) *endpoint* * ... * (α_n,β_n) *endpoint*

Disk Images as Typed Functions

```
val createOrderEntryRole : tPayment -> tOrderProcessing ->  (vm * tOrderEntry)
val createOrderProcessingRole : unit -> (vm * tOrderProcessing)
val createPaymentRole : unit -> (vm * tPayment)
```

```
let (vm1,e1) = createOrderProcessingRole ()
let (vm2,e2) = createPaymentRole ()
let (vm3,e3) = createOrderEntryRole e2 e1
```

- A value of type *vm* identifies a particular VM in the VMM
- VMs are booted from disk images held as physical files; the Baltic Server makes an RPC to the VM to set its imports (e2,e1) and get its exports (e3)

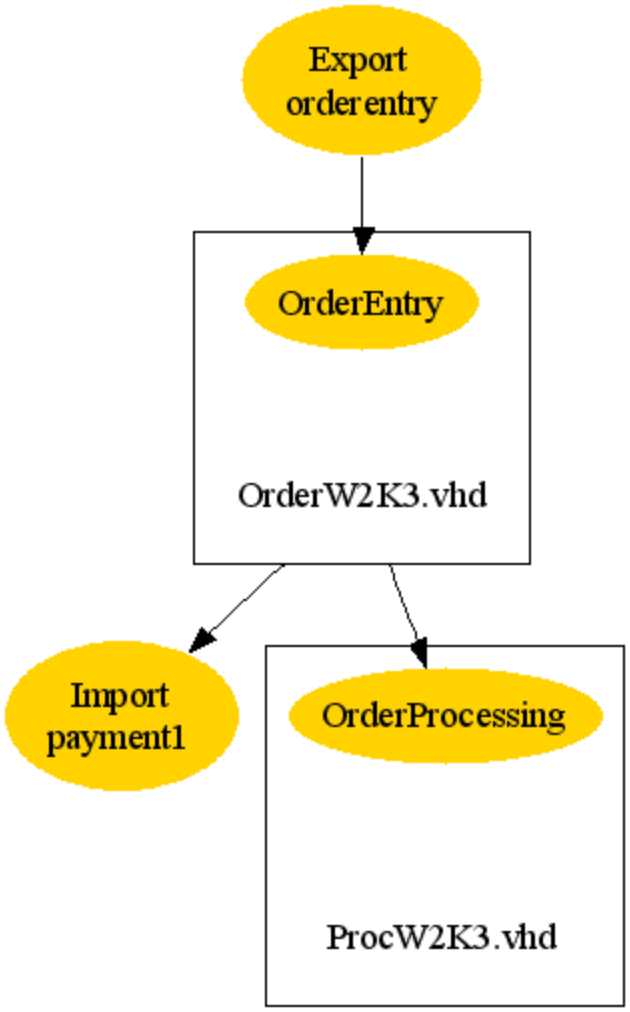


Import/Export as Typed Functions

```
val importPayment1 : unit -> tPayment
val importPayment2 : unit -> tPayment

val exportOrderEntry : tOrderEntry -> unit
```

```
let ei = importPayment1 ()
let (vm1,e1) = createOrderProcessingRole ()
let (vm2,e2) = createOrderEntryRole ei e1
let () = exportOrderEntry e2
```



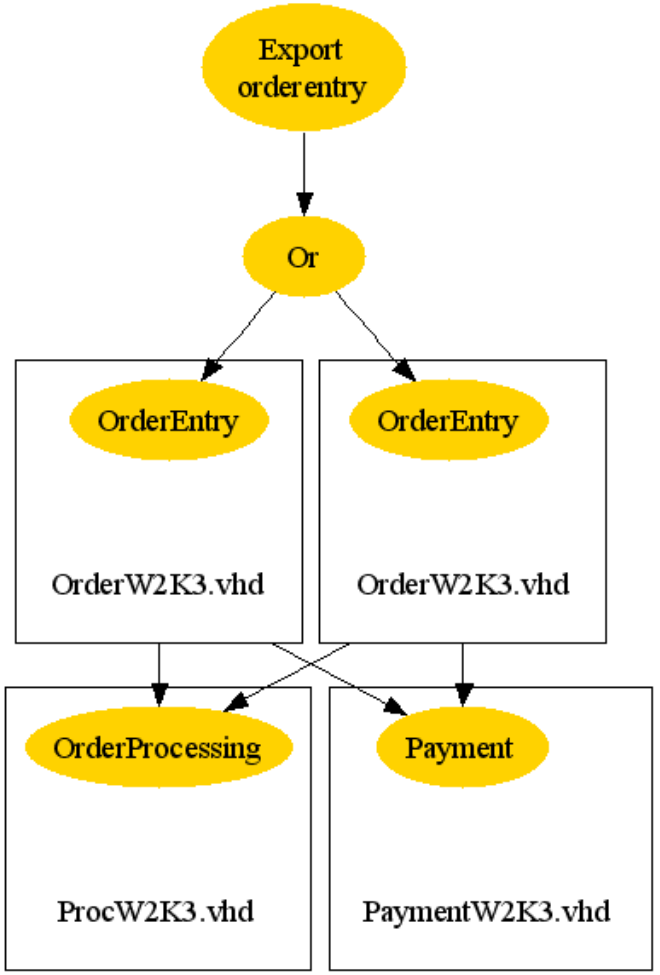
- Exporting or importing an endpoint creates a forwarder on the physical server
- The addressing details eg URLs are hidden by the API

An Intermediary for Load Balancing

val eOr : (α,β) endpoint -> (α,β) endpoint -> (α,β) endpoint

```
let (vm1,e1) = createOrderProcessingRole ()
let (vm2,e2) = createPaymentRole ()
let (vm3,e3) = createOrderEntryRole e2 e1
let (vm4,e4) = createOrderEntryRole e2 e1
let e34 = eOr e3 e4
let () = exportOrderEntry e34
```

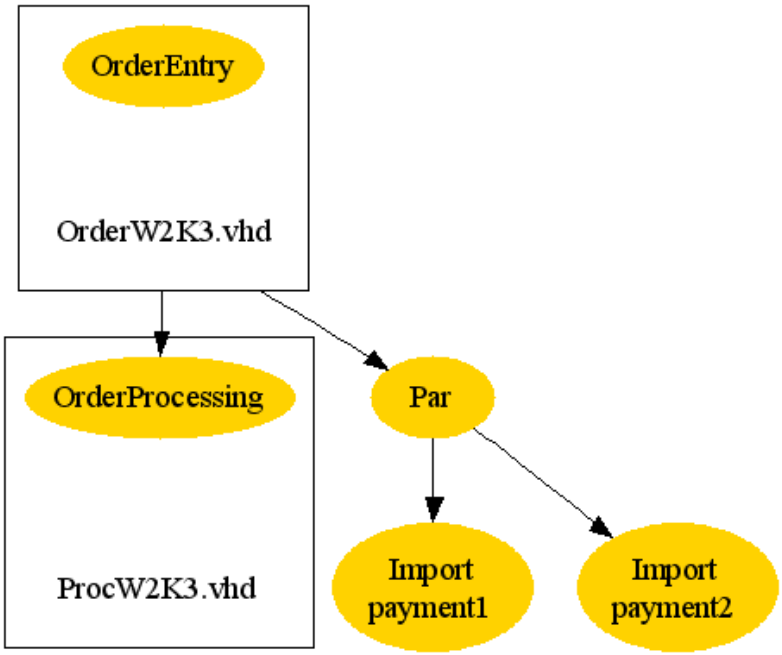
- We can utilize multiple processors by creating multiple instances of a role
- The Or-intermediary created by *eOr e3 e4* routes incoming requests to either *e3* or *e4*



An Intermediary for Fault Tolerance

val ePar: (α,β)endpoint -> (α,β)endpoint -> (α,β) endpoint

```
let e1 = importPayment1 ()
let e2 = importPayment2 ()
let e12 = ePar e1 e2
let (vm1,e3) = createOrderProcessingRole ()
let (vm3,e4) = createOrderEntryRole e12 e3
```

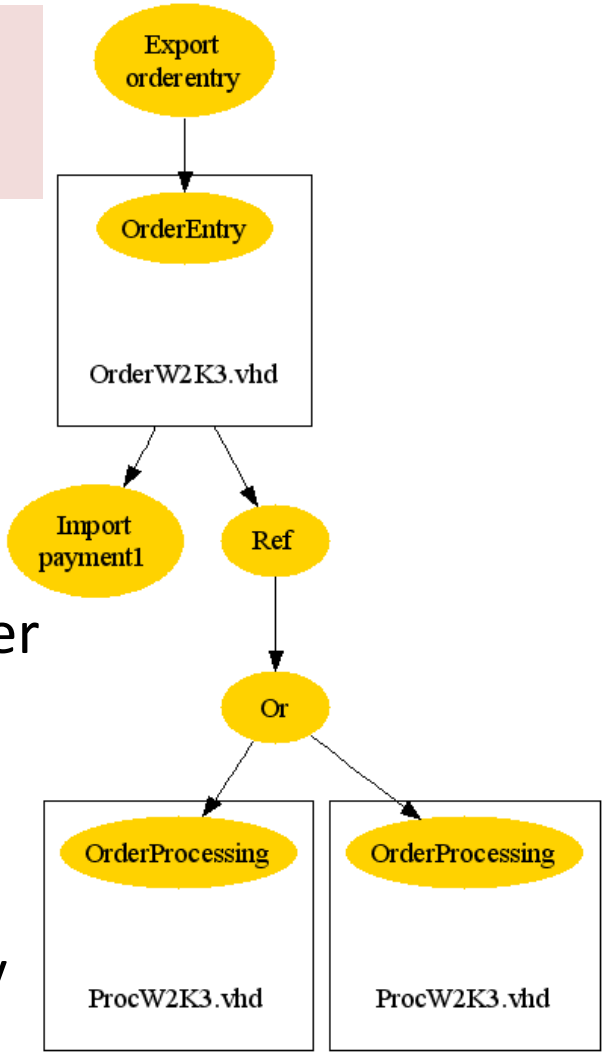


- The Par-intermediary created by *ePar e1 e2* routes incoming requests to both *e1* or e2; whichever returns first wins
- (Inspired by the service combinator *S1*|*S2* of Cardelli and Davies [1999])

An Intermediary for Dynamic Update

```
type (α,β) endpointref
val eRef : (α,β) endpoint -> (α,β )endpoint * (α,β) endpointref
val eRefUpdate : (α,β )endpointref -> (α,β) endpoint -> unit
```

```
...
let e3 = eOr e1 e2
let (e4,r) = eRef e3
let (vm5,e5) = createOrderEntryRole e0 e4
...
```

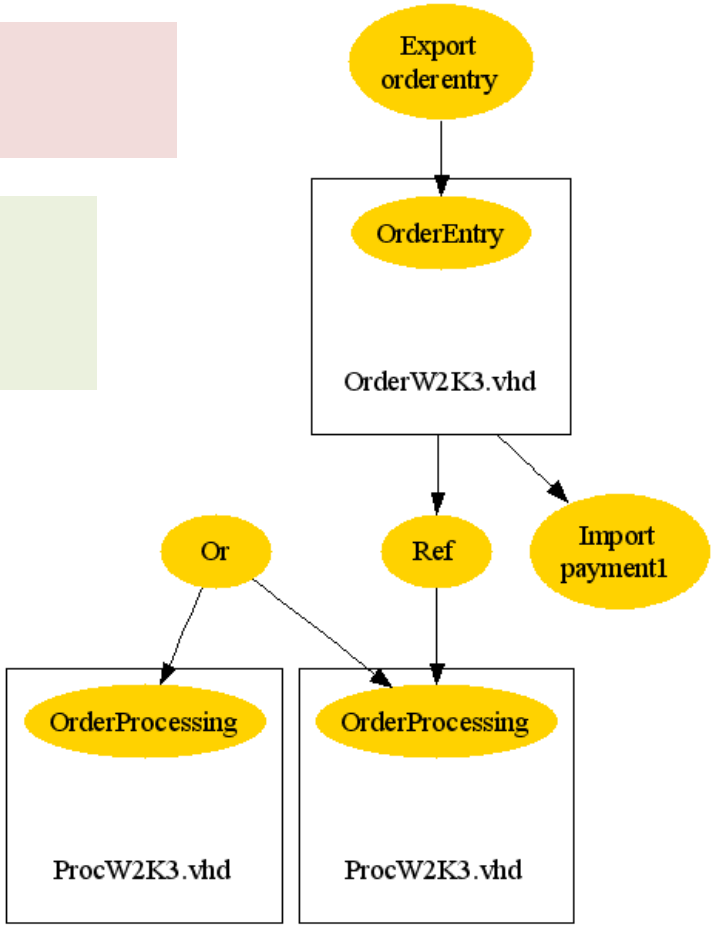


- A value of type *(α,β) endpointref* is a pointer to a Ref intermediary exporting an *(α,β) endpoint*
- The Ref-intermediary created by *eRef e3* forwards incoming requests to *e3*
- After *eRefUpdate r e'*, the Ref-intermediary forwards subsequent requests to *e'*

Event Handling

```
type event = VM_Crash
val eVM : vm -> (event -> unit) -> unit
```

```
let h e ev = match ev with VM_Crash -> eRefUpdate r e
let () =  Baltic.eVM vm1 (h e2)
let () =  Baltic.eVM vm2 (h e1)
```

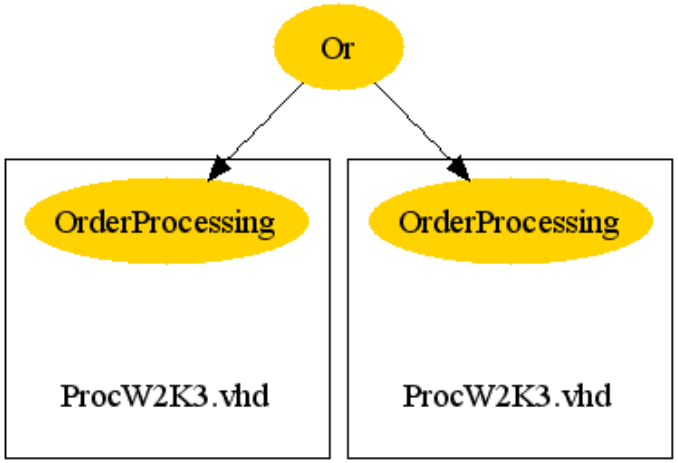


- VMMs detect many sorts of events, including VMs crashing
- *eVM vm h* installs *h* as a handler for events occurring on *vm*
- A current limitation is that we represent only the event *VM_Crash*

VM Snapshots

```
type vm_snapshot
val snapshotVM : vm -> vm_snapshot
val restoreVM : vm_snapshot -> unit
```

```
let svm1 = snapshotVM vm1
let svm2 = snapshotVM vm2
let h s ev = match ev with VM_Crash -> restoreVM s
let () =  eVM vm1 (h svm1)
let () =  eVM vm2 (h svm2)
```



- Some VMMs, including Virtual Server, allow snapshots of the current state of a VM and its virtual disks as physical files
- A value of type *vm_snapshot* is an identifier for such a state
- Using snapshots, we can revert a failed VM to a previous good state, faster than going through a reboot

MESSAGE SAFETY, TYPING, AND A FORMAL SEMANTICS

Baltic Scripts

- Given metadata *m*, let an *m*-script (a Baltic script) be a program that is well-typed given interfaces:
  - B.mli, the fixed part of the Baltic API; and
  - E*m*.mli, access to the roles and external endpoints in *m*.

- If we ensure Baltic scripts are well-formed by typechecking, we obtain a useful message safety property.
- An entity **respects** an endpoint of type (α,β) *endpoint* iff (1) each request sent by the entity to the endpoint has type α and (2) each response sent by the entity, in response to a request on the endpoint, has type β

Message Safety, by Typing

- **Message Safety Property (TBC)**: If
  - *m*-script S.ml passes typechecking, and
  - all remote entities respect the endpoints in *m*, and
  - disk images respect the endpoints they import and export

  then all entities arising during a run respect all endpoints.
- Many interconnection errors, where servers or intermediaries are connected to the wrong endpoints, lead to entities not respecting endpoints, that is, to requests or responses of unexpected types.
- These errors may arise at initial configuration, or sometime later, during reconnections.
- Our safety property guarantees, by static type-checking, that such errors cannot arise.

An Executable Formal Semantics

- Process calculi such as the pi-calculus are ideal for modelling highly-concurrent message-passing systems
- We build a process model of an *m*-script in two steps:
  - We implement the B.ml and E*m*.mli interfaces using concurrency primitives in an interface pi.mli, with (α,β) *endpoint* = $\alpha \rightarrow \beta$
  - We translate any program using just pi.mli into a process calculus (via a CPS transform followed by direct encoding)
- We prove that (1) well-typed programs map to well-typed processes, and (2) process computation preserves typings.
- Hence, we can formalize the type safety result (TBC!)
- To model multiple VMs, processes look like: $a_1[P_1] | ... | a_n[P_n]$
- To model snapshots, we allow partitions $[P]$ as values.

A Concurrent ML

pi.mli

```
type 'a chan                        // channel holding messages of type ‘a
type name                           // identifier
type part                           // partition [P]

val chan: string -> 'a chan         // create fresh channel
val send: 'a chan -> 'a -> unit     // send message on channel
val recv: 'a chan -> 'a             // receive message off channel
val fork: (unit -> unit) -> unit    // start thread in parallel
val name: string -> name            // create fresh name
val box: (unit -> unit) -> part     // create new partition
val start: name -> part -> unit     // start named partition
val stop: name -> part              // stop named partition
```

RELATED WORK AND CONCLUSIONS

Related Systems

- Edinburgh LCFG [Anderson 94] – scripting language to control large scale UNIX installations – shows farm management can be seen as a pure programming problem
- HP SmartFrog [Goldsack et al 03] – DSL to describe server components – typed, but not based on typed endpoints
- SDM/SML [MSFT 05] – XML formats for describing data centre components – not a programming language, but shows industry move towards explicit metadata for server farms
- Whitehorse [MSFT 05] – Distributed System Designers with both physical and logical views – visual notation, static topologies
- AppLogic [3Tera, 06] – AJAX visual designer for virtualized server farms – untyped scripting
- In the Singularity OS [Hunt, Larus, et al], typed message-passing is the sole means of interprocess communication. Disk images constructed using Singularity could be checked to conform to Baltic's assumptions

Conclusions

- We have described a set of combinators for assembling networks of virtual machines.
- A combination of type checking 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 pi-calculus with partitions and an implementation using Virtual Server with scripts in F#.

- Virtualization technology makes possible the automatic assembly of Big Components like DBs and OSs into systems
- An API like Baltic is a way to exploit this possibility in a typeful and declarative style

THE END