A Distributed Infrastructure for Mobile Computing

Andry Rakotonirainy - Martin Chilvers
DSTC
University of Queensland
Brisbane , Australia
{*andry,martin*}*@dstc.edu.au*

Abstract

The restrictions arising from the use of mobile components requires the design of a mobile infrastructure able to react to changes in the environment. We define an application aware adaptation architecture to cope with mobile computing constraints, where interacting objects use adaptation rules to sense and react to environment variations. This mechanism, provided by a mobile service, is built within an Open Distributed Environment (Hector) written in Python.

1 Introduction

The obvious characteristic of a mobile environment when compared to its stationary counterpart, is that it has dynamic constraints such as change of resources, removal of pieces of hardware, bandwidth fluctuations, limited memory storage, processing time, new environment configurations (a mobile object is migrated and wakes up in a totally different heterogeneous environment) and network disconnection/re-connection.
This work begins from the assumption that it is not possible to build abstractions or services that hide this underlying reality, if only because those constraints are physical. The management of mobile objects under these constraints implies that a set of events must be watched, managed or created; others might disappear and won't need further attention. Best effort is provided to react to and exploit fully the current environment to the benefit of interacting mobile and stationary objects. The main contributions of this work are the design of an architecture that provides the following functionality:

- An open distributed framework for mobile interactions based on the Hector[Arnold] platform. It includes binding, Quality of Service negotiation and design of adaptation rules that react to variations in the environment.
- A migration service: suspend, pickle, unpickle and resume any object from one name space to another and track active migrating objects.

Section 2 presents the Hector architecture. Section 3 shows how an interaction is processed in Hector, it describes binding, QoS negotiation and mobile adaptability rules. After that, section 4 defines the mobile service that implements the migration functionality. Section 5 is our conclusion.

2 Architecture

Our design is influenced by RM-ODP [Odp] and CORBA [Omg] to tackle openness, and it is built within the Hector [Arnold] platform. Hector is an object based distributed system platform. It is structured as four layered components representing decreasing levels of abstraction. These layers are the Object Layer, Language Layer, Encapsulation (or Kernel) Layer, and Communications Layer, as shown in Figure 1.
During execution, the three lower layers combine to form a single entity, called the "Capsule," that provides a support environment for distributed object execution. It encapsulates and insulates objects from the underlying machine and operating system architecture. A capsule defines interfaces to a set of fundamental services available to the objects that live "within". Object belonging to a capsule share the same address space (i.e. a capsule is a single process), with each object being allocated at least one thread.

2.1 Language Layer

The Hector kernel is designed to support objects written in different languages. A module that implements the mapping between the Capsule Kernel

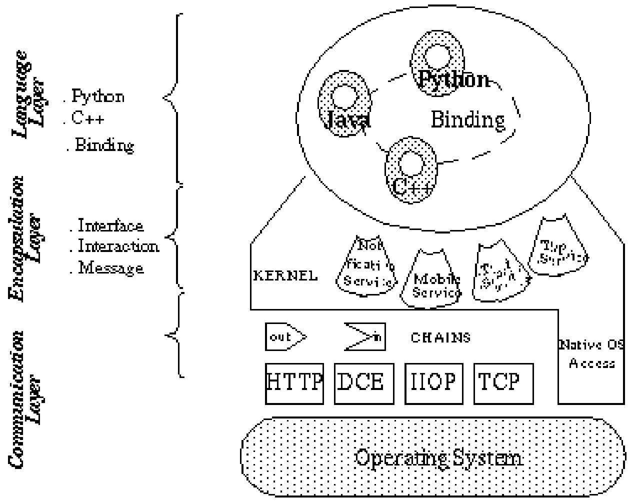

Figure 1: Hector Architecture

Layer (written in Python) and objects written in a particular language is called a "language mapping." The set of language mappings loaded into a Capsule forms the Language Layer.
Each language mapping performs two basic functions:

- Encapsulates the Kernel Layer classes, making them accessible from the object implementation language.
- Provides object factory functionality which consists of enabling (1) instantiation of objects using language-specific templates (2) execution of the instantiated objects.

The Language layer uses the notion of "binding" to allow interactions between distributed object's interfaces, and the RM-ODP [Odp] binding concept is central to our architecture. Binding templates define the interactions between interacting objects. Object interfaces take part in the binding by fulfilling a role. A binding's role is a placeholder for interfaces within a binding. It describes the required behaviour and others expectations of interfaces filling that role such as QoS. The notion of fulfilment is a subtype relation between role and interface. A binding can require that a role be optional and have cardinality constraints. Roles have a relationship such as "a client role needs a mobile agent role" between them. This concept is very important when describing mobile behaviour. A binding also contains a description of the interactions between its roles. In Hector, interfaces, roles and interactions are all described as input/output messages associated with a finite state machine describing the behaviour of those messages.Bindings are described as types and are stored in a repository server (type service). A binding does not make any assumptions about communication protocols nor the location of interfaces. Thus it uses different available services to provide transparencies (naming ...).

2.2 Encapsulation Layer

Encapsulation furnishes different types of services known as transparencies. We will describe only those services related to mobile computing management such as trading, naming, notification, and type management services.

- Event notification service: which allows an object to be aware asynchronously of the occurrence of a set of events from its environment by subscription. A callback function is triggered when the event is occurs. We assume that heterogeneous events of interest can be named and quantified in a domain. Our event notification service can be federated.
- Location service: trading and naming services are used to locate distributed objects. The database of these services are updated when the reference of an object changes.
- Type service: stores descriptions of types in a language independent way. Types can express behaviour descriptions. Such descriptions are used to instantiate objects in the language layer.
- Mobile service: this service exploits the above services and (1) drives adaptation rules (2) migrates objects (3) keeps track of moving objects.

2.3 Communication Layer

The communication layer provides transport-protocol transparency. The kernel is able to interact through multiple protocols while knowing nothing about the implementation of the transport protocol such as HTTP, IIOP or TCP. It also provides a chaining mechanism to keep track of moving objects using chain_in and chain_out functions on top of the transport protocol, as described in Section 2.3 .
By using a communication layer, a capsule can interact with other capsules. All types of faults occurring at this level are reported to the encapsulation layer using the event notification service. The encapsulation layer can directly access this layer, opening and closing connections via an abstract API.

3 How objects interact

In this section, we discuss the language layer abstractions that arise from the desire to have an application-aware adaptability mechanism that can copes with mobile constraints in an heterogeneous environment. A binding negotiation precedes any object interactions. This allows participants (objects) in the binding to negotiate QOS requirements, the type of the interaction, underlying protocols needed to implement the desired binding type. All these aspect are known as the binding type.

3.1 Binding negotiation

3.1.1 Binding type negotiation

A binding is a context for object interactions. A binding type, stored in a Type Service, describes how interactions between objects must occur. It also offers a domain which allows each interacting object to reach a common understanding of the semantics of all concepts used during interactions such as Integer, Names, QoS....
The initiator of the binding proposes a set of binding templates (with a set of interface references), of which one is eventually chosen once all mandatory roles are fulfilled by object interfaces. Apart from the pattern of interaction, the following aspects of computing must be also negotiated:

- ability to re-negotiate: which objects can initiate a re-negotiation, when and what is re-negotiable. What is the scope of a re-negotiation.
- ability to change adaptation (re-configuration) policy in terms of changing actions, conditions and events of interest. Those entities that have this ability must also be decided.
- cost of communication: this parameter is essential for mobile users
- security: security is crucial for mobile computers, the security requirement may depend on the location of the mobile host.

3.1.2 QoS negotiation

A binding template asserts that it performs properly if its interacting objects do. In order to guarantee that a binding will perform properly, binding roles must describe their QoS requirements, and interfaces must exhibit their capabilities as a promise to provide a level of QoS. QoS are specified as an attribute, behaviour or function. QoS is a part of the interface signature and can express multimedia attributes [Campbell].
Mobile components cannot guarantee QoS as stationary networks do [Abadi]. Nevertheless a mobile host can provide information about the environment in which it runs. An object runs in a specific type of environment that we call CoE (Class of Environment)[Andry]. The environment of a mobile object changes dynamically and corresponds to the quantified schema of configuration of local entities that it needs to process an interaction.
In order to tackle openness without losing flexibility, we classify objects in categories that reflect somewhat its CoE. The semantics of the classification are subject to negotiation and is defined within the binding domain. Entities in the classification range from battery power to screen definition through location of distributed objects. Concretely, CoE is created from a classification of the environment into tuples.
Example: one object can characterise and evaluate its environment along three axes:

- Memory/CPU/Graphics: This axis represents the Memory/CPU capability of a host. It is associated with the ability to support graphics because this depends on the memory/CPU capacity.
- bandwidth: Available communication bandwidth between interacting objects
- Disconnection: The rate of deliberate and accidental disconnection per period.

An interface with a CoE attribute can fulfil a role associated with a QoS if the mapping between the two concepts matches. The mapping is provided by the binding. It is therefore not recommended for an interface with a CoE characterised by high rate of disconnection to fulfil a role that participates in a multi party vote. Note that the binding might simply consider CoE as a QoS.

3.2 Interactions

Once the binding template is agreed, the binding and the object interfaces are instantiated. The binding will regulate interactions between fulfilled roles. Binding guarantees that interactions will eventually respect the behaviour described in the binding template.

3.2.1 Support for mobile constraints: adaptability

Interfaces may exhibit behaviour which allow them to adapt to mobility constraints. Our architecture supports the definition of adaptability rules at two levels:

- Directly by each interacting object: this complicates the design for object user-programmer. The infrastructure does not have to provide any mobile adaptation facilities
- Indirectly by the use of well known adaptation rules: This approach simplifies the task of the object user. The infrastructure provides facilities to manage mobile constraints.

These two approaches are semantically and syntactically similar.

3.3 Adaptation rules

Adaptation rules describe the policy used to react to changes in the environment. These rules range from a simple emission of a notification to complex actions such as re-negotiation or re-configuration of the system upon the occurrence of a critical event. It's a way to coordinate all actions necessary to cope with mobile computing

```
Example: if bandwidth < 9 Kbps do_something.
```

Adaptation rules are mainly specified by the application user using an ECA (Event Condition Action) structure. The Event Notification service is responsible for notifying the occurrence of events that an object has subscribed to.
The types of event/conditions can be: crossing a threshold, changes of configuration such as the migration of objects (instance), availability of memory, network disconnection, any types of fault. The types of actions are known as callbacks and can do nothing but notify the application object (exception), defer communications, keep track of instance of moving object, call appropriate functions, changing dynamically actions locally or remotely.
Event/conditions of an adaptation rule are expressed in a constraint language which provides a set of operators which allow arbitrarily complex expressions involving properties and values [Segall]. This description can be computationaly complete with respect to the observable properties of the local system if the programmer knows all observable events. An action is fired when the condition expression is true [Segall]. The following example contains four adaptation rules:

```
adaptation_rules = {
 "bandwidth > 20":increase_traffic_cb,
 "exists(tcp_errors)": use_udp,
 "cpu_load > 10": migrate_object_cb,
 "disconnected": re_negotiate_cb
}
```

Our adaptability rules syntax are generic enough so that we can express complicated policies such as priority of degradation for multimedia applications. For example, it allows us to specify which aspect of the multi-media stream we want to be degraded by priority when a range of events happen such as dropping the sound first then the number of frames per second).

4 Mobility Service

We have showed how our architecture copes with mobile constraints through the use of binding, QoS negotiation and adaptability rules. These are language level abstractions concepts. Engineering aspects require the collaboration of different services defined in the encapsulation layer. Coordination is provided by the mobile service
The support for mobility is build around the Event Notification Service [Segall](elvin) that make sure that the event condition part of adaptability rules are correctly checked and appropriate actions are triggered. The action part of adaptability rules might be provided by any available service such as the migration service, or directly provided by an interacting object.

4.1 Object migration

Migration [Hylton]is one type of action (amongst others) that can be specified in an adaptability rule. A migrating object can be either passive or active. A passive object is a class definition or a function (e.g.: a filter). An active object is an instance of the Hector object class. Both types of objects need to be accessed, tracked and instantiated from any capsule.The migration of an object is decided either by the mobility service or by the application object.

4.1.1 Moving an active objects

An active object migrates from a capsule to another capsule. Each capsule has a unique ID. The following migration class is inherited by all migratable objects.

```
import elvin # Notification service
class Migration:
 """Initialise an event notification and watch
 migrating events"""
 def __init__ (self):
   self.lv = elvin.elvin()
   self.lv.subscribe("exists(migrated)":\\
      chain_update)

 def move (self, obj, here, there):
   """suspend and copy the object"""
   p_obj= obj.suspend()
   send (p_obj, here, there)

 def suspend(self):
   """suspend and pickle an object
    with contexts"""
   T_id = get_reftype(obj)
   bind_contex = get_bind_context(obj)
   p_obj = pickle (obj, T_id, bind_context)
   notify("migrating", obj.__name__, there)

 def resume(self, p_obj):
   """retrieve instance of object"""
   obj, T_id, bind_contex = unpickle(p_obj)
   factory (Tid, obj, bind_context)
   notify("migrated", obj.__name__, there)

 def chain_update (self,d_not):
   """update reference"""
   if in_current_capsule(obj):
     -- if there is a chain_in pointer to this
     -- object in the capsule
     create_chain_out(obj, there)
     update (trader, name_service, obj,there)
```

Upon a decision to migrate an active object, the migration service does:

1. Suspend the execution of a running object. Get its type reference (Tid) description template from the type service so that the factory mechanism can have a description of this object at the other end. Pickle the object with information about the current context and the Tid. The context contains information about the current binding, the capsule, connections, and the current state. Notify the new location of the object to the event notification service so that those who are subscribed to migration events can be aware of.

2. creates a chaining mechanism "chain_update" to forward requests to the new location (chain_in, chain_out). Sends the object.

All capsules are notified about the migration of an object and their respective naming and trading services will update all relevant information about references of migrated object. This event flushes the service cache entries.
The receiving capsule will resume the object to retrieve ref_type and obj_state. The type service is used to retrieve the template with the ref_type. obj_state, ref_type and context that are used to run it in a specific environment.
When a migrated object finishes its execution or when the binding ends then all forwarded infrastructure is garbage collected.

4.2 Moving a passive object

The migration of a passive object is far simpler. Passive objects are either a class definition or a function. In Python, class definitions include the implementation of their methods. The type service does most of the work to achieve a class migration. Indeed, our type service is federated and stores a class definition so that a class can be instantiated from anywhere.
A function can be directly moved without informing the type service. A function will be evaluated by the receiving object. Such functionality allows us to the redefinition dynamic of the contents of a method or function within a targeted object. Such functionality is very useful in the following scenario: a video consumer with a poor CPU and screen capacity wants to receive a video from a producer. The consumer cannot exploit all of the data provided by the producer and wants to filter the stream. Thus, the consumer sends a filter (function) to the video producer based on its video definition support. The producer will send a filtered (reduced) amount of data thus sparing the bandwidth instead of having the consumer filtering incoming data and degrading its already poor CPU capacity.
In the Python world, before functions and classes can be migrated, they first have to be pickled. In standard Python, functions cannot be pickled at all (their code objects can be marshalled, but this does not provide sufficient information to allow a function to be re-created at a remote host), and pickled classes merely provide a reference to the module that they were imported from (meaning that source and target sites must either share a file system or at least mirror the same structure for module import purposes). To overcome these problems wecreate Python classes to represent fully-picklable functions and classes - FlatFunction and FlatClass (the implementation of FlatFunction is shown below). The

real work of these modules is provided by an extension module 'internals' that provides access to the Python C API (e.g. Py_FunctionNew). Now, given these modules we can pickle a class and its methods, and re-create that class at the target host even if the target host previously had no knowledge of the class (obviously, this raises some security concerns which are outside the scope of this paper).

```
FlatFunction.py
""" A class to represent fully 'picklable'
 functions. """
# Standard/built-in modules.
import marshal

# Common modules.
import internals

class FlatFunction:
""" A class to represent fully 'picklable'
functions. """
 def __init__(self, fn=None):
  """ Create a picklable representation
  of a function

  We use a default argument here so
   that the flattened function can be
   pickled/unpickled without the need
  for a '__getinitargs__' method.
   We do this so that the
  flattened function does not have
   to store any reference
  to the function object that
  it was created from."""

  # The only time 'None' is a valid
  #argument is when  the constructor
  #is being called by the 'pickle' module


  if fn != None:
    self._flatten(fn)
  return


def unflatten(self, namespace=None):
  """ Create a function object from the flattened
  function! """

  if namespace == None:
namespace = globals()

  return self._unflatten(namespace)


##########################
# Internal methods.
##########################
def _flatten(self, fn):
  """ Create a 'picklable' representation
  of a function."""

  # Get a list of the function's
  #default parameter  values, its
  #code object, and its documentation.

  self.defaults = fn.func_defaults
  self.code     = marshal.dumps(fn.func_code)
  self.doc      = fn.__doc__

  return


def _unflatten(self, namespace):
  """ Create a function object from
  the flattened function

  # Unmarshal the code object.
  code_object = marshal.loads(self.code)

  # Create a function object from the code object.
  function_object = internals.PyFunction_New \
 (code_object, namespace)

  # Set the default arguments.
  if self.defaults != No:
    internals.PyFunction_SetDefaults \
(function_object, self.defaults)

  # Set the documentation string.
  if self.doc != None:
    internals.PyFunction_SetDoc \
(function_object, self.doc)

  return function_object
```

4.2.1 References of mobile objects

The reference of an object is obtained by export and import to the trading or the naming service. There are two choices when handling references to active migrating objects. Either (1) each object currently interacting with the moving object updates their reference on the fly, or (2) they do not, and the capsule of the moving object has to provide an infrastructure to forward its communication to the new location. The first choice is expensive and might penalise all interacting objects involved in a multi party interaction. We take the second approach because the reference is transparently manipulated at the encapsulation language layer even if objects move during the life of a binding.

Intuitively, each interacting object is encapsulated in its capsule. A capsule has a reference to each interacting objects. These references are not necessarily location independent. The communication layer acts as an agent for the remote communication. It is responsible for stub generation and forwarding mechanisms. The stub contains the physical location of its interacting objects. Two functions, provided by the communication layer, are used to set up the tracking mechanism. These two functions are available to all migrating objects (by inheritance).They are:

- Chain_out: This function allows all request arriving on the former location of a migrating object to be forwarded to a new location.

- Chain_in:It allows a receiving capsule to forward a request to another capsule or to an object within this capsule. If the object is not located in the current capsule then chain_in will point to another chain_out functions forming a graph. At this stage we do not provide any mechanism for testing the acyclicity of a graph.

The forwarding mechanism and resources from original capsule are garbage collected when the binding completes.

5 Conclusion

We have presented an architecture for supporting mobile computing. Adaptability rules brings the awareness properties that allow an object to monitor resources and react to changes in its environment. It also provides the agility to metamorphosise according to the state of its environment. These rules are generic enough to cope with most mobile constraints. Rules can be customized to take appropriate actions when something bad such as disconnection occur. Binding concepts and QoS negotiation facilitate the design of an architecture, concept and algorithm to deal with mobile constraints.

6 Acknowledgments

The work reported in this paper has been funded in part by the Cooperative Research Centres Program through the department of the Prime Minister and Cabinet of the Commonwealth Government of Australia

References

[Abadi] M. Abadi, L. Lamport, *Composing Specification* , DEC - TR-66, October 1990 `http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-066.html`

[Arnold] D. Arnold, A. Bond, M. Chilvers *Distributed Object in Python* - Dr. Dobb's Sourcebook 273 January 1997. `http://www.dstc.edu.au/Hector/`

[Campbell] A. Campbell, G. Coulson, D. Hutchison , *A quality of service Architecture* - Journal of Computer Communication Review ,Vol 24 Number 2 April 1994.

[Odp] ITU/ISO , *Reference Model for Open Distributed Processing - Part 1,2,3 IS 10746-(1,2,3)*, `http://www.iso.ch:8000/RM-ODP/`

[Omg] Object Management Group - CORBAservices - , *Life Cycle Services Specification* , `http://www.omg.org/`

[Hylton] J. Hylton, K. Manheimer, F-L Drake Jnr, B. Warsaw, R. Masse, G. van Rossum , *Knowbot programming: System support for mobile agents* In Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96), pages 8-13,Oct. 1996.

[Andry] A.Rakotonirainy, A. Bond , *Mobile Transparency* , Proc ICODP, Toronto, May 1997. `http://www.dstc.edu.au/AU/staff/andry/icodp.ps`

[Segall] B. Segall *Elvin: Event notification service.* DSTC Internal Report, 1995. `http://www.dstc.edu/Elvin/`