Ponder2: A Policy System for Autonomous Pervasive Environments

Kevin Twidle, Naranker Dulay, Emil Lupu and Morris Sloman
Department of Computing, Imperial College London
{*kpt, nd, ecl1, mss*}*@doc.ic.ac.uk*

Abstract

Policies form an important part of management and can be an effective means of implementing self-adaptation in pervasive systems. Most policy-based systems focus on large-scale networks and distributed systems. Consequently, they are often fragmented, dependent on infrastructure and lacking flexibility and extensibility. This paper presents Ponder2, a novel policy system that is suitable for a wide range of environments and applications. The design and implementation of Ponder2 emphasises simplicity, flexibility and extensibility and provides users with the ability to interact easily with the managed system. Ponder2 can interact with other software and hardware components and is being used in environments ranging from single devices, to personal area networks, ad-hoc networks and distributed systems. We also describe PonderTalk, a high-level object orientated language inspired by Smalltalk for configuring and controlling Ponder2 systems.

1. Introduction

Pervasive Systems refer to a world where computational devices are embedded in the environment, worn by users or even implanted in their bodies. Applications range from body sensor networks for health monitoring [1], intelligent buildings, home automation [2], autonomous vehicles [3], and urban planning [3]. Such applications require continuous adaptation at the local device level as well as for collections of devices. The need for adaptation is driven not only by requirements changes but also by user mobility and context. Such devices have limited computational capabilities and strict power consumption requirements. Their operation must therefore be optimised and must constantly adapt in order to minimise resource consumption. Users are by and large not technically knowledgeable, and user interaction must be minimised.

Policy-based approaches are particularly suited to realising autonomous pervasive systems as they offer simple, flexible and dynamic technique for implementing adaptation and feed-back control. The management architecture in such systems must therefore be de-centralised and based on composition and peer-to-peer co-operation between autonomous elements.

Autonomic Computing is an initiative started by IBM [4] that is aimed at the more traditional management of large scale systems. It advocates a similar de-centralised model of autonomous agents co-operating with each other and composing into more complex configurations.

However, many existing policy-based frameworks have not been conceived for such environments. Their design is dependent on centralised infrastructure support such as LDAP directories and CIM repositories. Their deployment model is often based on centralised provisioning and decision-making that does not offer the means for policy execution components to interact with each other, collaborate or federate into larger structure. Policy specification is seen as an off-line activity, and policy frameworks do not easily interact with the managed system. Consequently such frameworks are difficult to install, run and experiment with. Additionally, they usually do not scale to smaller devices omnipresent in pervasive systems.

Ponder was a highly successful policy environment used by many in both industry and academia. Yet its traditional design suffered from the same disadvantages as described above. Whilst keeping some of the underlying concepts that have accounted for the popularity of Ponder we set-out to re-design the system entirely with the following goals:

Simplicity. The design of the system must be as simple and incorporate as few built-in elements as possible.

Extensibility. It must be possible to dynamically extend the policy system with new functionality, to interact with new infrastructure services and to manage new resources.

Self-containment. The policy environment must not rely on the existence of infrastructure services and must contain everything necessary to apply policies to managed resources.

Ease-of-use. The environment must facilitate the use of policies in new environments and the prototyping of new policy extensions for different applications.

Interactivity. It must be possible for managers and developers to simply interact with the policy system and the managed objects, issue commands to the managed objects and create new policies.

Scalability. The policy system must be executable on constrained resources such as PDAs, mobile phones and sensors, as well as more traditional distributed systems.

To this end Ponder2 comprises a general-purpose object management system with message passing between objects. It incorporates events and policies and implements a policy

execution engine. It has a high-level configuration and control language called PonderTalk while user-extensible managed objects programmed in Java are fully supported. In this paper we focus on obligation policies in the form of Event-Condition-Action rules authorisation policies has described in [5].

2. Self-Managed Cell (SMC)

Traditional approaches to policy based management have led to a number of languages including PCIM [6], PDL [7], NGOSS Policy [8], Ponder [9], and PMAC [10]. They all focus on either a condition-action or an event-condition-action rule paradigm. They either rely on a Policy Decision Point from which Policy Execution Points request policy decisions or on a deployment paradigm in which a deployment service pushes the policies to the agents that must enforce them. None of them provides the ability to dynamically interact with the environment in which policies are enforced, for example in order to issue management commands.

Ponder2 is implemented as a self-managed cell [11]. A self-managed cell is defined as a set of hardware and software components forming an administrative domain that is able to function autonomously and is capable of self-management. Management services interact with each other through asynchronous events propagated through a content-based event bus. Policies provide local closed-loop adaptation, managed objects generate events, policies respond and perform management activities on the same set of managed objects. An SMC can be thought of as the virtual-machine for Ponder2.

2.1. Managed Objects

Everything in Ponder2 is a Managed Object. Managed Objects include management policies and adaptors to real-world objects such as sensors, alarms, switches etc. Making everything a managed object allows Ponder2 to manipulate them all for management purposes in the same way. The basic Ponder2 objects include Events, Policies and Domains, it is up to the user to create or reuse Managed Objects for other purposes such as adaptors. Managed Objects, including all those mentioned, have to be loaded dynamically into the SMC thereby producing a factory managed object. Once loaded the factory managed object can be sent messages to create new instances of the desired managed object; these managed objects are the ones which do the work of the system. This is the same as any object oriented system where the class has to be loaded before instances can be created.

Once loaded, Ponder2 makes no distinction between factory managed objects and other managed objects. Both types can be sent messages asking them to do something and they both return replies. In the case of the factory managed objects they return a new instance of their underlying type.

2.2. Policies

Policies define the management goals of the system and events trigger reactions from the policies in order to deal with them.There are currently two basic policy types in Ponder2: Obligation Policies and Authorisation Policies.

Obligation Policies specify the actions that must be performed by managers within the system when certain events occur and provide the ability to respond to changing circumstances. For example, security management policies specify what actions must be performed when security violations occur and who or what must execute those actions. They can also specify what auditing and logging activities must be performed, when and by whom. Management policies could relate to management of QoS, storage systems, software configuration etc.

Obligation Policies are event triggered and are also known as Event Condition Action (ECA) policies i.e. they receive an event, check one or more conditions are true and if they all pass they perform one or more actions. The values held within the event may be used when evaluating the conditions and when executing the actions.

Authorisation Policies allow or deny message passing between managed objects. They define the activities a member of the subject domain (caller) can perform on the set of objects in the target domain (callee). These are essentially access control policies, to protect resources and services from unauthorised access. A positive authorisation policy defines the actions that subjects are permitted to perform on target objects. A negative authorisation policy specifies the actions that subjects are forbidden to perform on target objects. Authorisation policies are implemented on the target host by an access control component. Examples of policies are given in section 4.3.

2.3. Events

An event type is a managed object. An event type defines a type of event that may be produced some time in the future and whether any associated information is to be contained within the event. The event type is a template which contains the names of the arguments to be bound with the event when it is created.

An event is an instance of an event type and is produced by receiving a create message from a managed object within the system. The message may be sent as a result of a timer or a monitor detecting something or as the result of an external event. A managed object is always required to convert an external event into an internal event. An internal event may have values associated with it depending on the type of the event. Events are delivered to the event bus and thence to one or more policies. Events are also managed objects.

2.4. Domains

Domains are managed objects that act as containers for other managed objects in the same way that a directory or folder does in an hierarchical file system. The power of domains resides in the fact that policies in Ponder2 are normally specified in terms of domains, not on individual objects. [12]

3. Domain Event Bus

Ponder2 has an unusual graph-directed event mechanism. In a normal event bus, events are distributed to all objects that have registered to receive it, for example, to obligation policies that have specified that they want to receive a particular event. However, it is not always useful for all obligation policies to be activated by the same event so extra processing is required to determine if it is required or not. This results in extra complexity for the policy or a proliferation of event types to give a finer graduation of events.

Ponder2 solves this problem by the use of a Domain Event Bus which allows policies to pick up events generated only by certain managed objects. Essentially, instead of attaching policies to the event bus when they are activated, they may explicitly be attached to managed objects within the domain hierarchy. That is they subscribe to an event type via a managed object rather than via the global event bus. An event is always produced by a managed object somewhere within the system. Instead of the event being offered to all the policies in the system that are dependent on that event, the event propagates up the domain hierarchy. As it is propagated it is offered to any policies attached to any domain on the path up to the root as long as the event type is expected by those policies. In this way, a policy can be set up that will only respond to events created by a limited set of managed objects.

The more traditional Event Bus can be emulated simply by attaching all policies to the root domain meaning that all events are offered to all policies.

An example is shown in Figure 1. Two identical bed stations, each with a heart rate monitor which will produce a heart rate event should the heart rate cause concern. The bed policies and the ward policy have all subscribed to the heart rate event. With a traditional event bus, the bed policies would have to check where the event came from before proceeding. With the Domain Event Bus the bed policies will only see events generated from managed objects within their bed domains and the ward policy would see all events. The bed policies actions could involve just local managed objects and therefore they don't need any special knowledge about which bed they are monitoring. This allows more bed systems to be introduced within the ward without configuration and creating minimum impact on the rest of

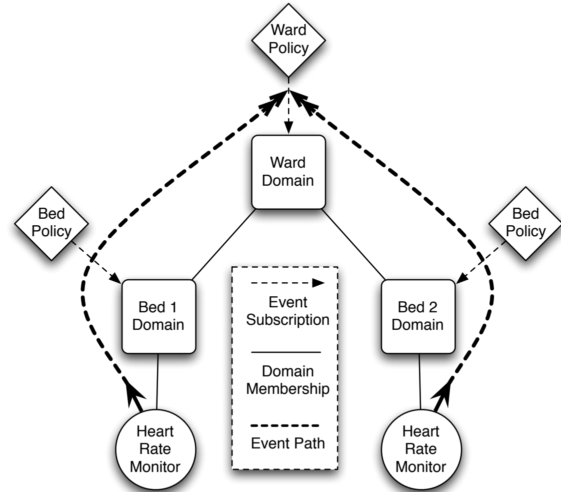

Figure 1. The Domain Event Bus

the system. The Domain Event Bus becomes even more relevant when applied to distributed and pervasive systems allowing local policies to deal with the local events while maintaining the ability to have higher-level policies reacting to the same event from many places.

4. PonderTalk

The first version of Ponder2 implemented the above object model but utilised XML to configure and control the policy system. This comprised of a `<use name=>` element identifying a path name to a managed object with child elements each representing a message to be sent to that object. This was simple and easy to implement and gave us experience but had two major drawbacks: One, XML does not make a nice programming language and it is hard to debug and read. Two, when writing a managed object in Java a great deal of code had to be written to decode the XML message that had been given to it; this could not be generalised because each object could receive arbitrary XML messages. Only after the XML message had been decoded could the real function of the managed object be invoked.

We needed a higher level language to configure and control Ponder2. This language had to be general but policy friendly and had to allow us to send messages to managed objects. We went through several design phases but could not come up with anything satisfactory and as flexible as the XML we were using. After thinking about the way that messages were sent to Managed Objects we realised that this was what Smalltalk [13] had been designed to do and decided to adapt it to our purposes. We do not have the concept of defining classes because objects are written in Java so we have utilised just the message passing aspects, some of the default types of Smalltalk and added a little to

the syntax and named it PonderTalk.

PonderTalk essentially identifies a managed object and then sends it messages with optional arguments. PonderTalk has the concept of variables that can be used in place of path names to identify managed objects. Commands are of the form:

```
pathname message(s).
myVar := pathname.
myVar message(s).
```

A parser grammar for PonderTalk was written using the ANTLR [14] parser generator. The parser generator compiles PonderTalk into XML which is executed at runtime. This new XML contains type information about parameters allowing calls to be made directly to the users' Java methods (see section 4.4) obviating the need to decode the XML and to removing the need for basic type checking.

4.1. Booting

When the Ponder2 is started, it only contains a root domain, nothing else at all. Not even the domain factory exists to allow other domains to be created. Since this is probably not what is wanted, a default PonderTalk bootstrap script is read and executed to load a bare minimum of useful managed objects. The root domain is a special domain and has a few extra messages it can receive, one of these being the import message to load a factory managed object. The object loaded is actually a Java class picked from the class path.

4.2. Blocks

Blocks are a very powerful feature of PonderTalk. They are closures and can be viewed as creating dynamic functions. A block in PonderTalk is enclosed in square brackets [and]. It contains arguments and statements e.g.

```
[:name :age | root print:
  ( name + " is " + age + " years old").]
```

When the above construct is compiled and executed it returns a block managed object which contains the argument names and the compiled code as an XML structure. This block managed object (referred to as a block) can be treated in the same way as any other managed object. It can be assigned to a variable, added to a domain, and sent messages. In this case the block receives messages with argument values, telling it to execute the compiled code within the block with the given arguments.

Block arguments are actually treated as variables. Use of arguments in the block's statements look exactly the same as using variables to refer to objects. Blocks form closures, so when a block is created it maintains a copy of the current variables. When a block is executed the arguments are handed to the block as part of the execution message. The arguments are simply added to the block's set of variables and the xml code is executed. The XML code itself is exactly the same as any other PonderTalk XML and can now be executed in the same way. The fact that the block's variable table has had addition(s) does not matter since the next time the block is executed the same variable(s) will be overwritten by the new arguments.

4.3. Policies

The policies described in section 2.2 are imported, created and used in the same was as any user written managed object but have special functions within the Ponder2 system when they are activated.

Obligation policies are sent messages to set up the event, conditions and the actions. The conditions and actions are in fact PonderTalk blocks that the policy executes when necessary. When activated, the policy is attached to the domain event bus and is handed events by the internal event mechanism whenever the appropriate event is created. When the policy receives an event, it executes any condition blocks by sending them the event as a message. If the result is true then the policy executes the action blocks in a similar manner. The blocks, when executed, take their argument values from the event by name. If, for some reason the event does not have the appropriate values, an error is thrown.

```
Policy := root/factory/ecapolicy create.
Policy event: myEvent;
       condition: [ :arg | bool-expression ];
       action: [:arg | statements ]
```

Authorisation Policies authorise the sending and receiving of messages. These policies have source and target objects and the operation that is to be permitted or denied. In terms of a distributed system they govern the message leaving the source object, the message arriving at the target object, the reply leaving the target object and the reply arriving at the source object. In the following example nurses in ward 1 are given permission to read ward 1's patients' records. The *focus* tells the policy that it is protecting the target object(s).

```
Policy :=  (root/factory/authpolicy
  subject: root/personnel/nurse/ward1
  action: "getrecord"
  target: root/patient/ward1
  focus:"t" .
```

4.4. Java Managed Objects

The functionality of Ponder2 systems is greatly enhanced by the introduction of user-written Managed Objects. Java was chosen as it is widely known and has an extensive system library available. To facilitate this support is provided for programmers to write Managed Objects that accept

messages from other Managed Objects, including the PonderTalk interpreter. An ideal solution would be to create automatically some mapping code that takes the PonderTalk messages and arguments, organises the argument types and calls a Java method within the managed object. This would mean creating some stub code at compile time.

Java annotations of the form `@annotation(arguments)` are used. When the compiler come across such an annotation, user-level factory code is called with the structure of the class or method associated with the annotation being made available by the Java compiler. In this way PonderTalk message names are mapped by annotations to methods within the Managed Object. The annotations are placed above Java methods and are made available to compiler extensions at compile time. The compiler extensions have full access to the method parameter types enabling stub code to be generated to perform mapping between the generic message format and the Strings and ints etc. required by the method in question. For example to create a Managed Object that accepts the `at:put:` keyword message that stores a name with an integer value e.g. `myObj at: max put: 1000` we can write the following method:

```
@Ponder2op("at:put:")
public void store(String name, int value) {
...
}
```

To be a managed object the object must implement an empty interface called ManagedObject. This interface simply tells the Java compiler that this is will be a Ponder2 Managed Object and it must start creating Java adaptor code to perform the requisite mappings from PonderTalk to Java methods.

If we want to send a message to a managed object, for example, to set the name and age of a person this can be done in PonderTalk as:

```
myobject name: Fred age: 24 .
```

the Java code would simply be:

```
@Ponder2op("name:age:")
public void setInfo(String name, int age){
  this.name = name;
  this.age = age;
}
```

4.5. Ponder2 Interactive Shell

In addition to managed object support, Ponder2 contains an internal, interactive command program, rather like the Unix shell program. The Ponder2 shell is primarily a debugging and testing aid and is often used for browsing the domain structure of a Ponder2 system. The Ponder2 shell has some internal commands similar to Unix commands for moving around and listing the contents of the domain structure, it also accepts and executes PonderTalk statements.

5. Inter-SMC Interaction

Ponder2 SMCs interact with remote Managed Objects as though they were local. This is totally transparent to the messaging system and PonderTalk. We require two things to be able to communicate in this way: a reference to the remote object and some form of communication medium.

All Managed Objects in Ponder2 have a unique reference identifier whether they are local or remote. Remote references are managed by a special External Managed Object which acts as a local proxy for the remote object. The proxy transparently marshals and unmarshals messages as necessary. As far as Managed Objects and PonderTalk in the local SMC are concerned they are simply dealing with a local object.

Objects are imported into an SMC using the root domain's import: command. The argument being a URL referring to the remote SMC and the path name of the object within it. The returned reference creates a proxy External Managed Object within the SMC and that is what is returned to the caller.

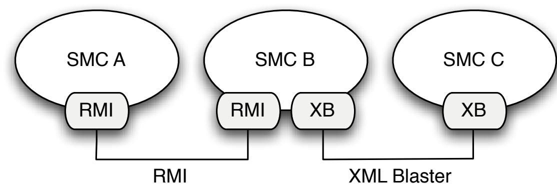

Figure 2. Protocol Plug-ins

In Figure 2 we can see three Ponder2 SMCs A, B and C. Each SMC can be started with the appropriate -address argument giving it its own address where it will listen for messages. The following communication command arguments will give us this configuration :

```
SMC A: -address rmi://SMCA
SMC B: -address rmi://SMCB -address xb://SMCB
SMC C: -address xb://SMCC
```

Given this startup example SMC A can now import a domain from SMC B and similarly SMC B can import a domain from SMC C

```
// SMC A
root/A1 at: "B1"
   put: (root import: "root" from: "rmi://SMCB").

// SMC B
root at: "C1"
   put: (root import: "root" from: "xb://SMCC").
```

After these import statements have been executed, SMC A will see something like Figure 3 in its domain hierarchy. Managed Object A can communicate freely with any object within SMC A e.g. `root/A1` using internal Java calling mechanisms and with objects in SMC B e.g. `root/A1/B1/B2` using the RMI protocol and with objects

within SMC C e.g. `root/A1/B1/C1/MO_C` using the XB protocol. Note that Managed Object A has no knowledge of the protocols being used when sending a message, it simply addresses the other objects using their pathnames.

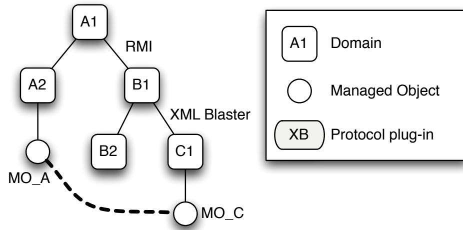

Figure 3. Imported Domain Hierarchy

6. Conclusions and Future Work

The Ponder2 system is proving to be very versatile and has been applied in many projects both at Imperial College and by other institutions [15], [16]. Ponder2 has been utilised on robots, body sensor nodes and mobile telephones. Research projects include health monitoring using body-area networks of sensors and actuators [11], unmanned autonomous vehicles [17] as well as large web-service based infrastructures [18]. The software, documentation and tutorials are available from www.ponder2.net.

There is on-going work on the design of a policy-specific language which will have tighter constraints on its expressions and actions; much like the original Ponder policy specification language. This will allow formal analysis, such as conflict detection, to be performed across a set of policies which is an almost impossible task with a general programming language such as PonderTalk.

Acknowledgment

We gratefully acknowledge financial support of the CISCO Research Center Grant - DIPOLE: Distributed Policy Execution.

References

[1] G.-Z. Yang, Ed., *Body Sensor Networks*. Springer, 2006.

[2] C. Nugent and J. Augusto, Eds., *Smart Homes and Beyond: Proc. 4th Int. Conf. on Smart Homes and Health Telematics*. IOS Press, 2006.

[3] The Cityware Project. [Online]. Available: http://www.cityware.org.uk/

[4] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[5] G. Russello, C. Dong, and N. Dulay, "Authorisation and conflict resolution for hierarchical domains," in *8th IEEE Int. Workshop on Policies for Distributed Systems and Networks(POLICY)*, Bologna, Italy, 2007, pp. 201–210.

[6] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. (2001) Policy core information model – version 1 specification. [Online]. Available: http://www.faqs.org/rfcs/rfc3060.html

[7] J. Lobo, R. Bhatia, and S. A. Naqvi, "A policy description language," in *Proc. 16th Nat. Conf. on Artificial Intelligence AAAI/IAAI*, Orlando, Florida, USA, 1999, pp. 291–298.

[8] J. Strassner, *Policy-Based Network Management: Solutions for the Next Generation*. Morgan Kaufmann, 2003.

[9] N. Damianou, N. Dulay, E. Lupu, and M. Sloman, "The Ponder Policy Specification Language," in *Int. Workshop on Policies for Distributed Systems and Networks (POLICY)*, vol. 1995, Bristrol, England, 2001, pp. 18–39. [Online]. Available: http://pubs.doc.ic.ac.uk/Policy2001/

[10] D. Agrawal, S. B. Calo, J. Giles, K.-W. Lee, and D. C. Verma, "Policy management for networked systems and applications," in *IFIP/IEEE Symp. on Integrated Network Management*, Nice, France, 2005, pp. 455–468.

[11] E. Lupu, N. Dulay, M. Sloman, J. S. Sventek, S. Heeps, S. Strowes, K. P. Twidle, S. L. Keoh, and A. E. S. Filho, "Amuse: autonomic management of ubiquitous e-health systems," *Concurrency and Computation: Practice and Experience*, vol. 20, no. 3, pp. 277–295, 2008.

[12] M. Sloman and K. Twidle, "Domains: A framework for structuring management policy," in *Network and Distributed Systems Management*. Addison Wesley, 1994, ch. 16, pp. 433–453.

[13] A. Goldberg and D. Robson, *Smalltalk 80: The Language*. Addison-Wesley, 1989.

[14] T. Parr, *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[15] H. Zhao, J. Lobo, and S. M. Bellovin, "An algebra for integration and analysis of ponder2 policies." in *POLICY*. IEEE Computer Society, 2008, pp. 74–77. [Online]. Available: http://dblp.uni-trier.de/db/conf/policy/policy2008.html#ZhaoLB08

[16] R. Neisse, P. D. Costa, M. Wegdam, and M. van Sinderen, "An information model and architecture for context-aware management domains." in *POLICY*. IEEE Computer Society, 2008, pp. 162–169. [Online]. Available: http://dblp.uni-trier.de/db/conf/policy/policy2008.html#NeisseCWS08

[17] E. Asmare and M. Sloman, "Self-management framework for unmanned autonomous vehicles." in *AIMS*, ser. Lecture Notes in Computer Science, A. K. Bandara and M. Burgess, Eds., vol. 4543. Springer, 2007, pp. 164–167. [Online]. Available: http://dblp.uni-trier.de/db/conf/aims/aims2007.html#AsmareS07

[18] The TrustCoM project. [Online]. Available: http://www.eu-trustcom.com