Presented at IFIP/IEEE International Symposium on Integrated Network Management
(ISINM 95), Santa Barbara, USA, May 1995, published by Chapman & Hall, pp 29-42

Configuration Management For Distributed Software Services

S. Crane, N. Dulay, H. Fosså, J. Kramer, J. Magee, M. Sloman, K. Twidle
Imperial College, Department of Computing, London SW7 2BZ.
E-mail: mss@doc.ic.ac.uk

Abstract

The paper describes the SysMan approach to interactive configuration management of distributed software components (objects). Domains are used to group objects to apply policy and for convenient naming of objects. Configuration Management involves using a domain browser to locate relevant objects within the domain service; creating new objects which form a distributed service; allocating these objects to physical nodes in the system and binding the interfaces of the objects to each other and to existing services. Dynamic reconfiguration of the objects forming a service can be accomplished using this tool. Authorisation policies specify which domains are accessible by which managers and which interfaces can be bound together.

Keywords

Domains, object creation, object binding, object allocation, graphical management interface.

1 INTRODUCTION

The object-oriented approach brings considerable benefits to the design and implementation of software for distributed systems (Kramer 1992). Configuring object-structured software into distributed applications or services entails specifying the required object instances, bindings between their interfaces, bindings to external required services, and allocating objects to physical nodes. Large distributed systems (e.g., telecommunications, multi-media or banking applications) introduce additional configuration management problems. These systems cannot be completely shut down for reconfiguration but must be dynamically reconfigured while the system is in operation. There is a further need to access and reconfigure resources and services controlled by different organisations. These systems are too large and complex to be managed by a single human manager. Consequently, we require the ability not only to partition configuration responsibility within an organisation's managers but also to permit controlled access to limited configuration capabilities by managers in different organisations.

This paper describes the SysMan configuration management facilities for open distributed software services. We use the Darwin notation to define the *structure* of a distributed service or application as a *composite object* type which defines internal primitive or composite object instances and interface bindings (Magee 1994). The external view of a service is in terms of interfaces *required* by clients and *provided* by servers. Managed objects implement one or more management interfaces providing management services and event notifications to managers. In the following we use the terms ‘object reference’ interchangeably with ‘interface reference’ since an object is uniquely identified by one of its interface references.

A domain-based infrastructure is used to group object references. This can be used to partition management responsibility by grouping those objects for which a manager is responsible. Furthermore, domains provide naming contexts in which interfaces are registered. (An interface can be included in more than one domain.) The domain service thus performs two functions: it associates management policy with groups of objects and it permits managers to associate convenient names or icons with interface references.

A graphical user interface permits a human manager to locate managed objects by browsing through the domain hierarchy. Once located, composite objects may be inspected and their internal configuration of interconnected object instances modified. New applications can be constructed by interactively creating object instances and binding their interfaces to those already registered in the domain service. Figure 1.1 shows the overall environment. A manager locates interfaces in the domain service via a configuration manager object (CM) and invokes operations on these interfaces to create or delete objects, bind interfaces or perform application-specific management.

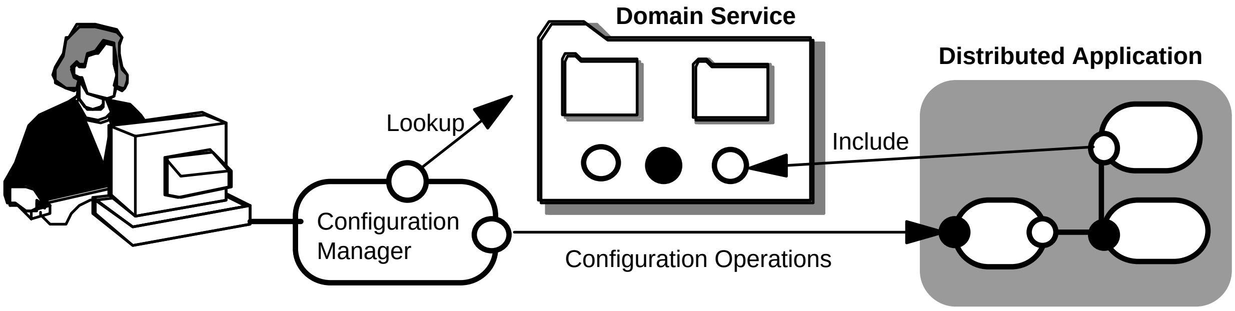


Figure 1.1 Interactive configuration management.

The term ‘configuration management’ often connotes those activities concerned with setting internal object state, for example: updating routing tables, adjusting numbers of buffers and specifying device addresses. We assume that these functions are performed by invoking operations on objects and use the term to describe the management of the structure of objects constituting a distributed service.

In section 2 we give an overview of the use of domains in the SysMan management environment and then in section 3 we use the Active Badge Location Service as an example to describe the configuration facilities of the Darwin Language. In section 4, we discuss issues relating to creating objects followed by binding of interfaces in section 5. The user interface for configuration management is described in section 6 and is followed by related work and conclusions.

2 MANAGEMENT ENVIRONMENT

2.1 Domains and Policies

Domains provide a means of grouping object interface references and specifying a common policy which applies to the objects in the domain (Sloman 1989, 1994, Moffett 1993, Twidle 1993). A reference is given a local name within a domain and an icon may also be associated with it. If a domain holds a reference to an object, the object is said to be a *direct member* of that domain and the domain is said to be its *parent*. A domain may be a member of another domain and is then said to be a *subdomain*. Policies which apply to a parent domain normally propagate to subdomains under it.

An object (or subdomain) can be included in multiple domains (with different local names in each domain) and so can have multiple parents. The domain hierarchy is not a tree but an arbitrary graph. An object's direct and indirect parents form an *ancestor* hierarchy and a domain's direct and indirect subdomains form a *descendant* hierarchy (Figure 2.1). The domain service supports operations to create and delete domains, include and remove objects, list domain members, query objects' parent sets and translate between path names and object references (Becker 1993).

An authorisation policy is specified by an *access rule* which defines a relationship between managers (in a subject domain) and managed objects (in a target domain) in terms of the management operations permitted on objects of a specific type (Moffett 1993, 1994). Policies applying to a user or manager are defined in terms of a *User Representation Domain* (URD)**,** a persistent representation of that person in the domain system. When they log into the system, a CM object is created and included in the URD. Policies specified for their URD then apply to their CM object.

2.2 Domain Browser

The *Domain Browser* is a graphical interface common to all management applications (Sloman 1993). It permits a human manager to navigate the domain structure; select objects and include or remove them from domains and invoke operations on selected objects. The browser displays tree diagrams with ancestors in the left window, the current domain in the middle and descendants to the right, Figure 2.1. The current domain, `hal,` has two direct parents: `/home` and `/users/staff`, and itself contains two domains: `mydomain` and `tmp`. It is possible to indicate cycles and collapse parts of the tree (not shown in Figure 2.1). A displayed domain can be selected to become the current domain in the window or a new window can be opened.

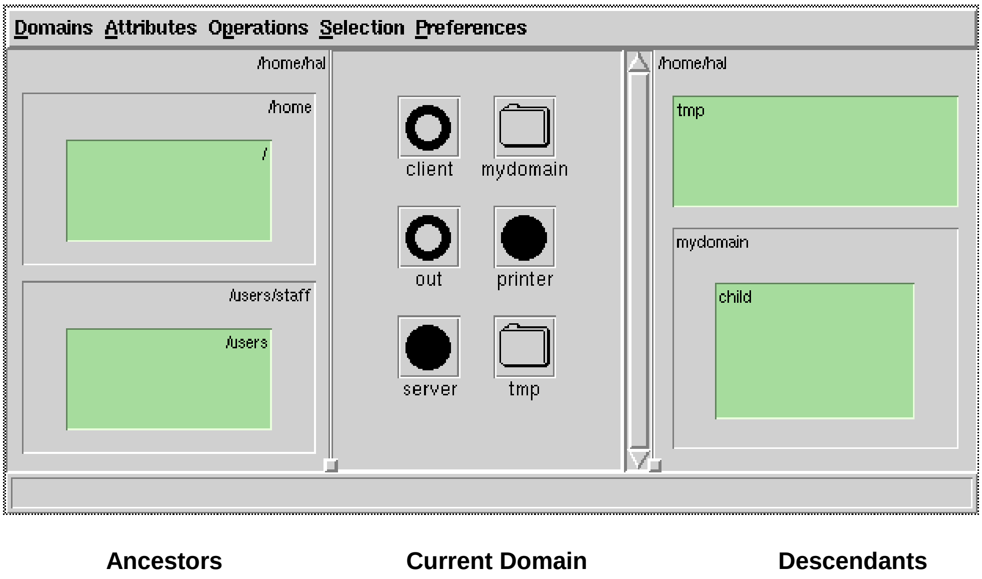


Figure 2.1 Domain window with hierarchy views.

Directories in the UNIX file system can also be displayed as domains via an adapter object included in a domain. (However, it is not possible to include files into domains or object references into a UNIX directory.) The domain browser is used to navigate the file system and select an object template (stored as a program file) which can then be used to create object instances (described further in section 4).

2.3 Operation Invocation

The following attributes are associated with an interface reference present in a domain:

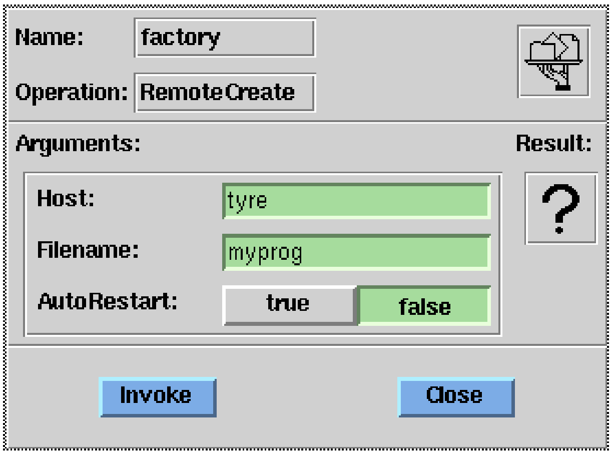
Local name: a textual name which uniquely identifies the interface within the domain.

Object identifier: a unique identifier used to invoke operations on the interface.

Icon reference: specifies its appearance in the graphical interface.

Type reference: used to query a type store for the interface's operation signatures.

The type information associated with an object specifies the operations which can be invoked on the object and the parameters they require. Operations are invoked on an object from the Domain Browser by selecting the object icon in the current domain window then selecting an operation from a pull down menu which lists the names of the operations supported by the object's interface. The Domain Browser uses the operation name and associated type information to generate a dialogue box for the user to supply required arguments, Figure 2.2.



Name: factory

Operation: RemoteCreate

Arguments:

Host: tyre

Filename: myprog

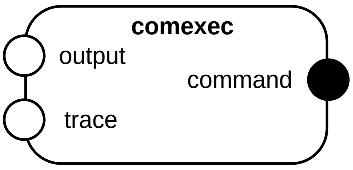
AutoRestart: true [✓ false]

Figure 2.2 Dialogue box to invoke operation with parameters.

The user enters parameters for the invocation in the dialogue box and presses Invoke. The user interface performs the invocation, updating the dialogue box with the result. The domain browser also supports *drag-and-drop* invocation; selecting an icon in one domain and dropping it onto another invokes the `include` operation on the destination domain.

3 ACTIVE BADGE LOCATION SERVICE

Examples in this paper are taken from an Active Badge system implemented using the SysMan environment. Active Badges (Harter 1994) emit and receive infrared signals which are received and transmitted by a network of infrared sensors connected to workstations. Badges can be worn by people or attached to equipment. The system permits the location and paging of badges within range of a sensor.

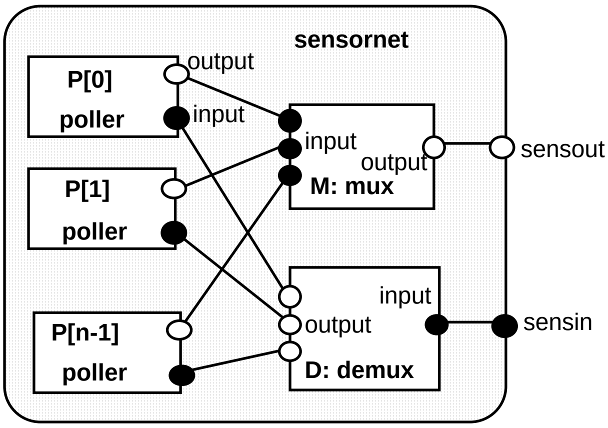


```
component comexec {
  require  trace   <event bstatus>;
           output  <port   smsg>;
  provide command <entry comT repT>;
}
```

Figure 3.1 Component type.

The object in Figure 3.1 provides a service via an interface (depicted by a filled circle) but requires two external services (empty circles). It executes badge commands to set off a badge's internal beeper or to illuminate status LED's. By convention, the first word of the type specification (in angle brackets) is the *interaction mechanism* class. For example, `command` accepts 'entry' calls with a request of type `comT` and a reply of `repT`. To execute a command, it is first necessary to locate a badge. Consequently, `comexec` requires the `trace` service which gets location events of type `bstatus` from an event service. The component sends a message to the sensor network to transmit the command to the badge, once found via `output` which possesses 'port' semantics.

Composite distributed services are constructed by composing object instances, Figure 3.2. The `sensornet` component controls access to the sensor network. Each requirement (empty circle) in this example is for a port (`output`) to which messages are sent, and each provision (filled circle) is a port (`input`) on which messages are received. Internal interfaces can be made visible at a higher level by binding them to the composite component interface, e.g. `M.output` is bound to `sensout` and `sensin` to `D.input`.



```
component sensornet(int n) {
  provide sensin    <port smsg>;
  require  sensout <port smsg>;

  array P[n]:poller;
  inst
     M:mux;
     D:demux;
  forall i:0..n-1 {
     inst P[i] @ i+1;
     bind
         Pli].output -- M.input[i];
         D.output[i] -- P[i].input;
  }
  bind
    M.output -- sensout;
    sensin     -- D.input;
}
```

Figure 3.2 Composite component type.

Each poller component is located on a different workstation and controls a multidrop RS232 line of sensors. It requires a service to output badge location sightings and provides a service on which it transmits commands. In general, many requirements may be bound to a single provided interface; however, in this case, each poller instance's `output` is bound to a separate `input` port to allow the multiplexor `M` to identify the particular poller `P[i]` from which a message is received. Pollers are distributed by the expression `inst P[i]@ i+1` to locate each instance (`P[i]`) on a separate machine (`i+1`). Machine identifiers are mapped to physical machines at run time which permits a configuration specification to be reused in different environments.

The `sensornet` component of Figure 3.2 forms a subcomponent of the badge manager, `badgeman`, Figure 3.3. This server provides the following interfaces:

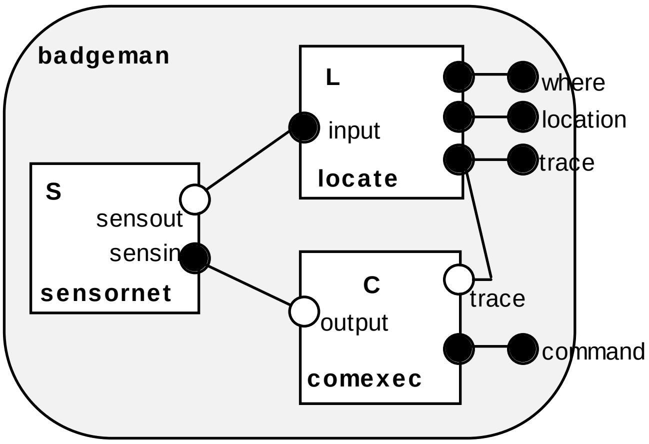
`where` to query the locations of all badges,

`location` to receive all location-change events,

`trace` to receive location change events for a particular badge,

`command` to execute a command on a badge.

When `badgeman` is created, it registers these interfaces in the domain 'badge' (which is assumed to exist). Darwin's `export` statement indicates that the reference to a provided service interface should be registered externally. Conversely, an `import` statement allows required services to be found in the domain service.



```
component badgeman {
 export
   where      @ "badge/where",
   location   @ "badge/location",
   trace      @ "badge/trace",
   command @ "badge/command";
 inst
   S: sensornet(4);
   L: locate;
   C: comexec;
 bind
   where -- L.where;
   location -- L.location;
   trace -- L.trace;
   command -- C.command;
   S.sensout -- L.input;
   C.output -- S.sensin;
   C.trace -- L.trace;
}
```

Figure 3.3 Exporting services to the domain service.

In practice, on-line configuration of the badge system is desirable (for example to add new pollers as the sensor network is extended). In the following, we demonstrate how the composite service of Figure 3.2 may be represented in Darwin to permit dynamic configuration.

4 OBJECT CREATION

As we have seen, an object can contain multiple composite or primitive objects, distributed over many nodes. It can export multiple service interfaces which can be included in domains to permit binding. In this section we describe management facilities supporting object creation.

Local Creation Service (LCS)

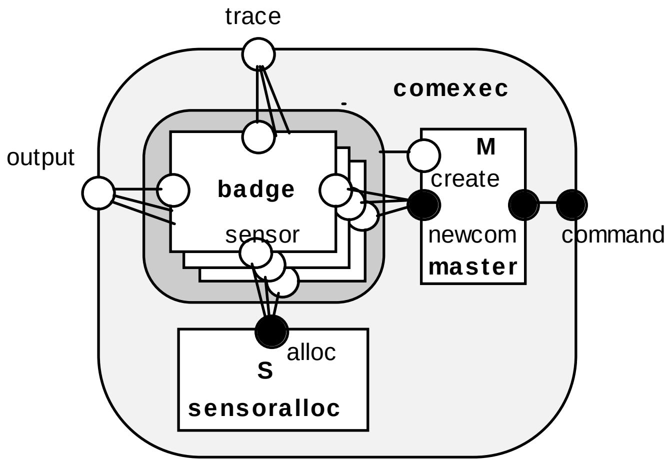
This is provided by the operating system. For example, the badge server can be created simply by executing a command from a UNIX shell. Once executing, its interfaces appear in the domain ‘badge’. (This implies that the operating system, which is outside of the domain service context, must be able to include interfaces in a domain**.**)

Remote Creation Service (RCS)

The example in Figure 3.2 requires a remote creation service to instantiate a poller at a node different to that of the multiplexor and demultiplexor. This service creates distributed objects by providing access to the LCS on a remote node, (Crane 1994).

Internal (Darwin) Creation Service

A Darwin program may create objects statically at the time the composite is instantiated or dynamically using the keyword `dyn`. New objects may be instantiated entirely within an existing object or they may make use of the LCS or RCS to create composite objects on new nodes. In Figure 4.1, `master` dynamically creates a badge proxy to handle each request for command execution.



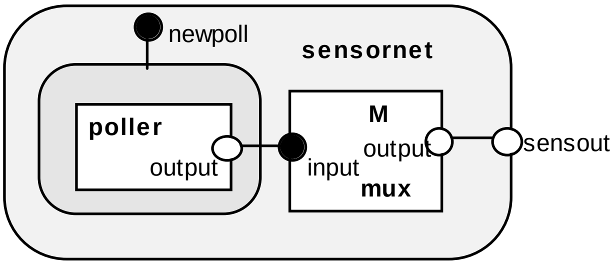
```
component comexec {
  require  trace   <event bstatus>;
           output   <port  smsg>;
  provide  command <port comT>;

  inst
      M :master;
      S :sensoralloc;
  bind
      M.create -- dyn badge;
      badge.trace -- trace;
      badge.sensor  -- S.alloc;
      badge.output  -- output;
      badge.command --M.newcom;
      command -- M.command;
}
```

Figure 4.1 Dynamic object instantiation.

Application-Provided Creation Service

An application interface may provide a specific operation to create objects in the context of the composite object. For example, Figure 4.2 depicts a simplified version of Figure 3.2 in which poller objects can be added by invoking the `newpoll` service. It uses a different poller object taking a single parameter which determines its location (c.f. Figure 3.2).



```
component sensornet {
  require sensout<port smsg>;
  export  newpoll <dyn int>
    @ "badge_admin/newpoll";
  inst
    M: mux;
    D: demux;
  bind
    M.output -- sensout;
    poller.output -- M.input;
    newpoll -- dyn poller;
}
```

Figure 4.2 Dynamic object instantiation service.

Interactive Object Creation

The configuration manager permits a human manager to access all creation services via a graphical interface which is described in section 6. Figure 4.3 indicates how this service uses the other creation services.

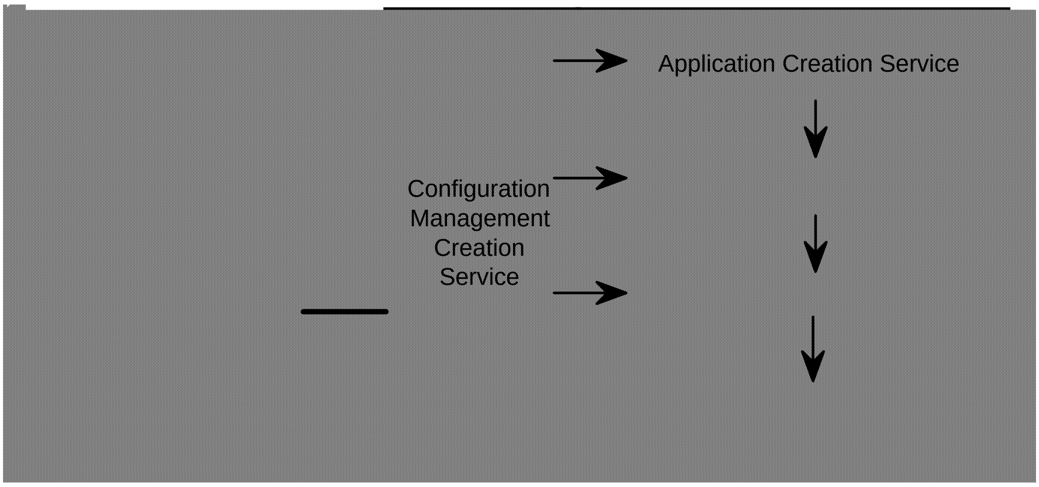


Figure 4.3 Creation mechanism relationships.

5 OBJECT BINDING

A required interface must be bound to a provided interface before a client can invoke operations on a server. There are two fundamental binding operations:

Binding create a link between a required interface on a client and a provided interface on a server using an external 'third-party'.

Unbinding destroy an existing binding.

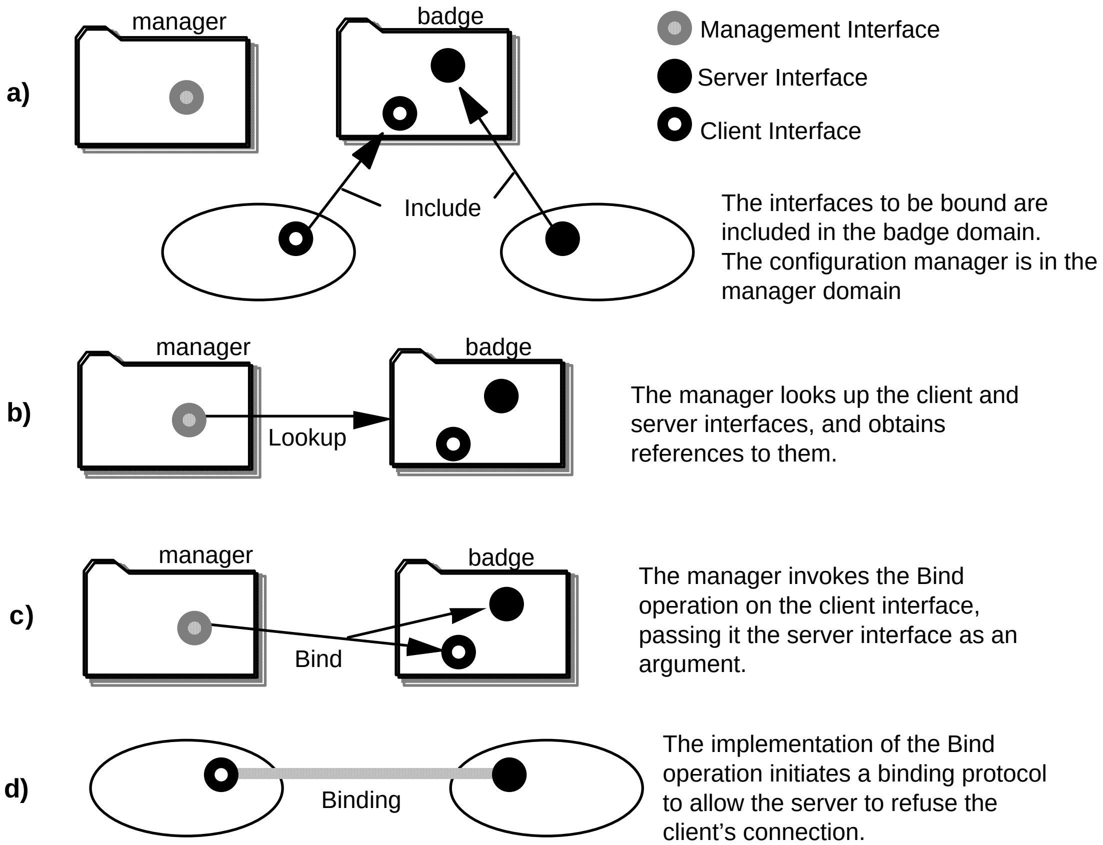
Rebinding is performed by first unbinding and then binding. Destroying a running object instance will generally require its interfaces to be first unbound.

Whereas it may be assumed that unbound program components are in a consistent state prior to binding, this is certainly not always the case before unbinding. Therefore a protocol is needed for 'safe' unbinding and rebinding. It will be explained in section 5.4.

5.1 Third-Party Binding

In the examples of sections 3 and 4, bindings are performed by an external third party (the manager) or they are defined by a Darwin configuration. Objects being bound do not play an active part in binding; they are unaware of the interfaces to which they are bound. The advantage of this approach is that structure is defined explicitly rather than being hidden in an object's internal state. Figure 5.1 shows the stages in the interaction of a configuration manager with the domain service to locate and bind interfaces.

This example requires certain access rules to be present: `manager` requires 'lookup', 'bind from' and 'bind to' permission on `badge`. An additional access rule specifies the operations the client can invoke at the server interface.



The interfaces to be bound are included in the badge domain. The configuration manager is in the manager domain

The manager looks up the client and server interfaces, and obtains references to them.

The manager invokes the Bind operation on the client interface, passing it the server interface as an argument.

The implementation of the Bind operation initiates a binding protocol to allow the server to refuse the client's connection.

Figure 5.1 Third party binding interactions.

5.2 First-Party Binding

Many distributed systems, e.g. (ANSAware 1993), make use of first-party bindings in which a client locates a server using a name server and establishes the binding itself. This type of binding is very common in open systems: a client can locate the services it requires with no intervention by a manager. It assumes that the information which enables the client to find the required service (a name or service description) is compiled into the client or passed as an instantiation parameter.

The Darwin language also permits first-party bindings to be specified for a composite object:

```
component view (int dt) {
        require  locations<entry int statT>;
}

component where (int dt=0) {
        import locations @ “badges/where”;
        inst v: view (dt);
        bind v.locations -- locations;

}
```

This shows a client of the badge manager which polls the latter's `where` service periodically (or once if no parameter is given). It queries the badge domain for the `where` service, gives it an internal name (`locations`) and binds its internal interface to this service. The client requires an access rule permitting 'lookup' and the server requires an access rule permitting 'include' on the domain `badges/where`.

5.3 Dynamic Invocation Bindings

A third type of binding arises when a reference to a *provided* interface is passed in a message to a client, which implicitly assigns it to a *required* interface, and uses it to invoke operations on the provided interface. This mechanism is suitable for dynamic environments which cannot afford the overhead of either first- or third-party bindings.

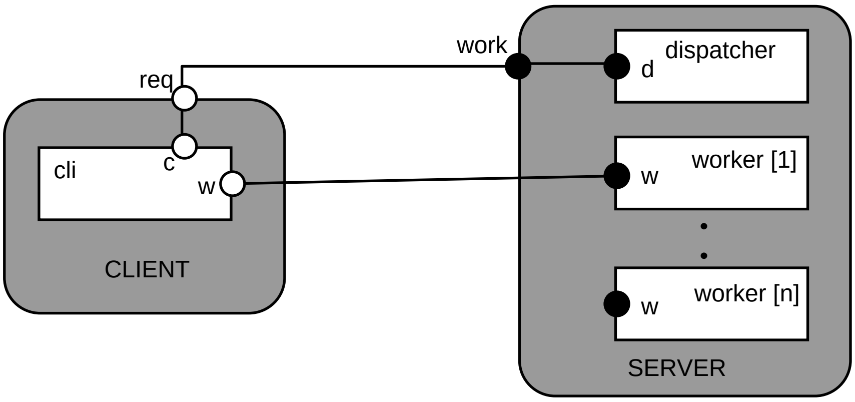


Figure 5.2 Dynamic binding example.

In Figure 5.2, the client's `req` interface reference is initially bound to the server's `work` interface. To access one of the server's worker processes, the client sends a request and receives a reply containing a reference to a worker's interfaces (at `dispatcher`'s discretion) which it assigns to `w`. Communication between `cli` and `worker` then proceeds independently of dispatcher.

An access rule is required to permit the binding between `cli` and `worker`, but this can only be checked when `cli` invokes an operation on `worker` (unless a bind protocol has previously been executed).

5.4 Safe vs. Unsafe Unbinding

Destroying a binding is more complicated than creating one, because it might be in use at the time of removal. Bindings are part of an application's overall state, and applications normally require *safe* unbinding, requiring a consistent state to be reached before bindings are destroyed (Kramer 1990). A request for immediate unbinding is usually unsafe (but perhaps desirable in certain circumstances).

In general, safe unbinding entails the co-operation of the programmer. Our approach requires programmers to mark bindings *critical* in sections of code where unbinding would cause inconsistency. When bindings may be safely removed, they are marked safe. If an unbind request arrives when an interface is critical, it is blocked until the binding becomes safe.

In the Regis system (Magee et.al. 1994, Crane 1994), many communication styles are available. The simplest and most flexible of these is the *message port*, but programmers must explicitly render them safe for unbinding. Regis also provides objects similar to Ada's *entries* which have semantics similar to RPCs. These are safe to reconfigure as long as no calls are outstanding on them, which can be determined by the support system. Another communication object, providing an even more rigid style of communication, is the *event distributor* used in the badge system. For these objects, safety is synonymous with the desire to receive event notifications; when enabled, they are *critical*, when disabled they may be safely rebound. (An attempt to transmit on an unbound interface will block the transmitting process until binding occurs.)

6 INTERACTIVE CONFIGURATION MANAGEMENT

As mentioned in section 1, the configuration manager (CM) supports Domain browser facilities to locate interfaces, and functions to display composite object structure and invoke operations on interfaces from within a ‘configuration window’, as will be explained in section 6.2. The CM permits a user to associate an invocation signature with an interface and to specify an icon to represent it. For example, to create an object at a particular node, the CM is used to locate an object type in the file system which is then dropped onto the required node icon in the RCS domain, Figure 6.1.

6.1 Configurable Composite Objects

The configuration management view of a distributed application is an extension of the domain browser view. A user employs the domain browser to navigate to a composite object. A composite object with visible structure is represented by a *Configuration Domain* which displays internal interfaces as icons (Figure 6.1). This *domain view* is similar to an ordinary domain but it is not possible to include external objects *into* a configuration domain although objects can be included *from* a configuration domain *into* other domains. A configuration domain can optionally display a *structural view* showing bindings between internal interfaces, permitting a manager to monitor the system structure and make changes to it. There may not be a complete view of all internal interfaces, but only the rebindable ones on which configuration operations are possible. The configuration domain is effectively a management interface to a composite object and is included in a domain when the object is created. Objects visible in a configuration domain may themselves be configurable composite objects.

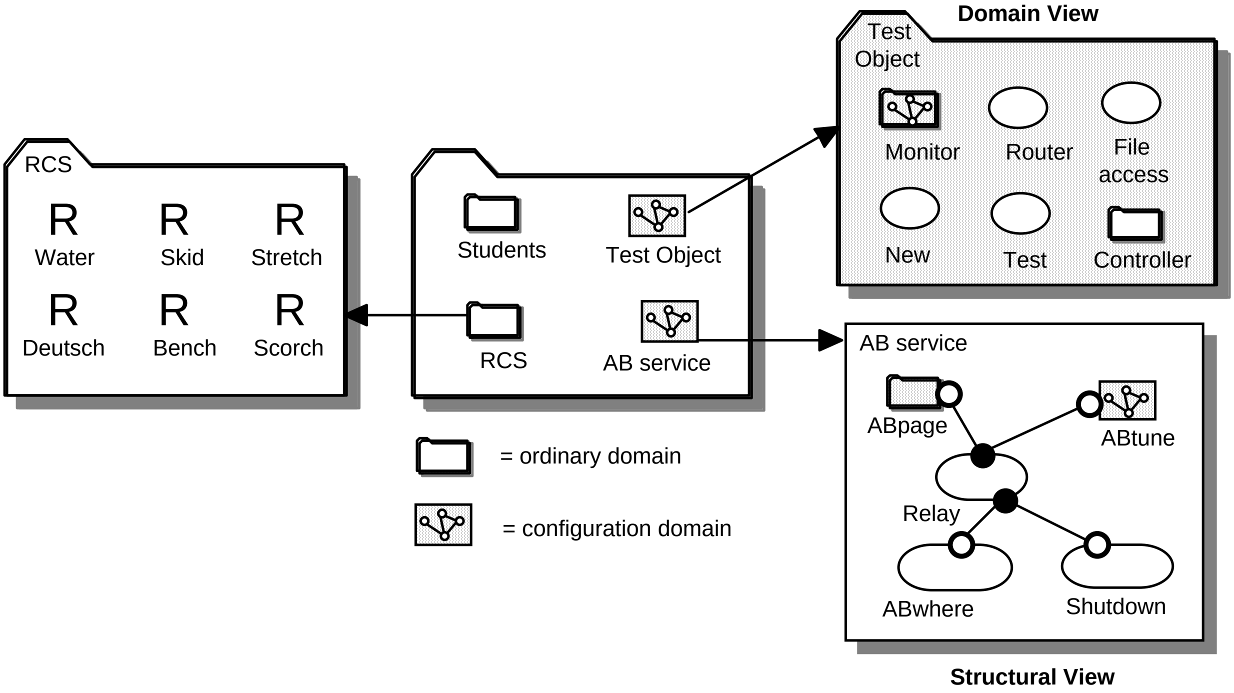


Figure 6.1 Domains and services with special and default icons.

6.2 Configuration Window

A user of the CM performs interactive binding in a *configuration window* which displays a structural view of a configuration.

Figure 6.2 shows how `Relay` in the `AB Service` configuration window can be bound to `New Relay` in domain `Test` by a drag-and-drop operation. The drop invokes the Bind operation on the target, and results in the configuration window being updated to show the new binding.

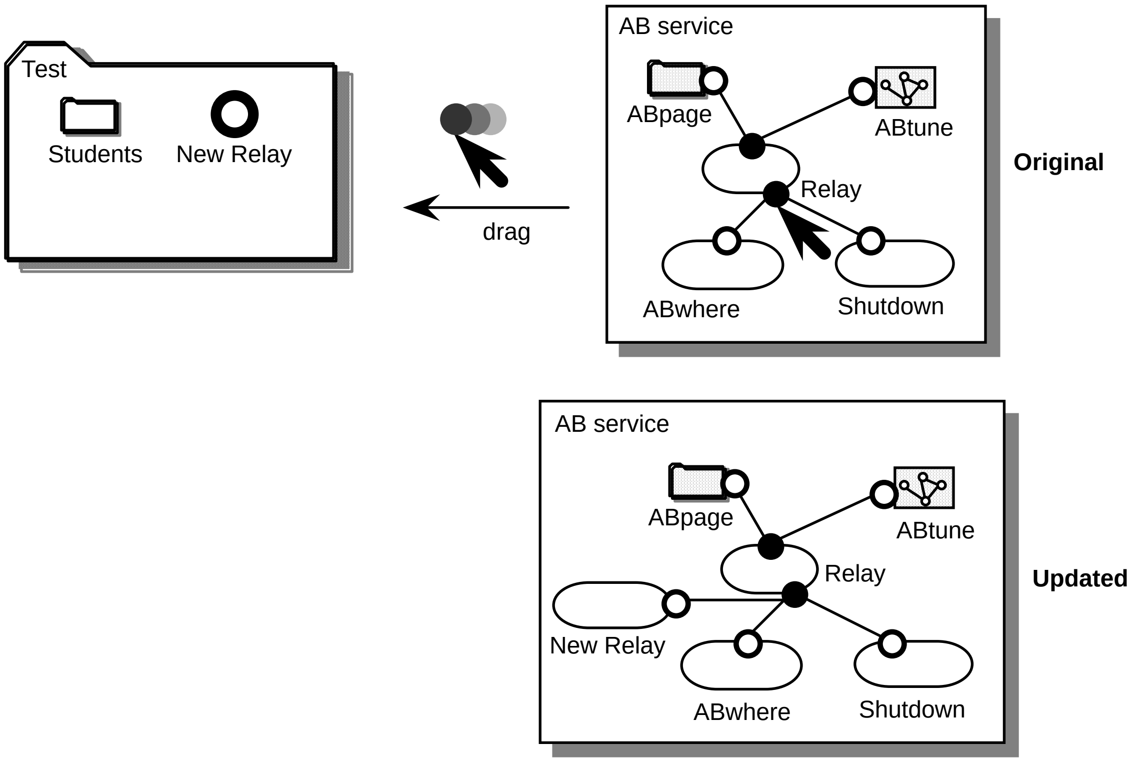


Figure 6.2 Drag-and-drop binding.

6.3 Current Status

The domain browser, object invocation via dialogue windows and structural views of configuration domains have been implemented and drag-and-drop interactions are being implemented. The Darwin compiler works in the Regis programming environment and has been modified to support ANSAware objects. The RCS allows creation of distributed objects defined by a Darwin program.

7 CONCLUSIONS AND RELATED WORK

This paper has shown how a graphical interactive configuration management facility can be used to manage software objects comprising a distributed application or service. Our approach has evolved over many years of experience with Conic, REX, and Darwin which have been used by industrial and academic institutions.

The use of directories in name servers to hold references to objects is common in distributed systems (Leser 1993), but domains extend this concept to applying policies to contained objects. The naming provided by domain path names is for user convenience rather than to provide a unique name for an object. DEC also use the concept of domains to group objects for management purposes (Strutt 1991) and the Ansa Trader uses domains as a trading context (ANSAware 1993). Our approach goes further than trading in that it shows how to use domains for interactive configuration management.

A number of other systems provide a configuration language (Agnew 1994, Zimmermann

1994, Barbacci 1993), which have some similarities to Darwin but our approach is the only one to combine *static* initial configurations, *dynamic preplanned* reconfigurations and *evolutionary* or unplanned dynamic reconfigurations. We cater for configuration management of both ‘closed’ systems (i.e. single applications) and ‘open’ systems consisting of multiple applications bound using the domain service. However, further work is needed to gain more experience with the current models for safe reconfiguration (Kramer 1990) and some of the more restrictive but practical proposals (Agnew 1994).

Key concepts in our approach are:

- *Explicit structure*. Both the Darwin notation and the graphical configuration view explicitly identify software structure in terms of object instances and interface bindings. A graphical tool, capable of generating Darwin code, allows design of composite components by stepwise refinement (Kramer 1993).

- *First- and third-party binding*. First-party binding is useful in some circumstances. However object-oriented systems which only support first-party binding often require binding information to be embedded in clients. This makes reuse difficult. Third-party binding permits structural information to be defined at the configuration level, resulting in configurations which are ‘cleaner’ and easier to understand.

- *Hierarchical Composition*. The ability to create composite services interactively *and* within the Darwin language provides a very powerful way to generate new services from existing services either statically or dynamically.

- *Evolution* is supported at two levels: pre-programmed change can be incorporated in composite objects using the dynamic facilities of the Darwin language, while interactive configuration management facilities are used to introduce new types and replace existing ones with minimal interruption of service.

- *Domains* provide a means to group interfaces and partition the overall management of the system by representing organisational or physical structure. Combined with access rules they provide scope for specifying policies relating managers to managed objects.

- The *domain browser* allows management interfaces to be located and draws upon the experience of file system user interfaces found in many operating systems.

8 ACKNOWLEDGEMENTS

The authors acknowledge the support of the Commission of the European Union through Esprit project 7026 (SysMan) and DTI support of Eureka project IED 4/410/36/002 (ESF). We acknowledge the contribution of our colleague Keng Ng to the concepts described in this paper.

9 REFERENCES

Agnew B., Hofmeister C., Purtilo J. (1994) Planning for Change: a Reconfiguration Language for Distributed Systems, In IOP/IEE/BCS *Distributed Systems Engineering*, **1:5**, 313–322.

ANSAware (1993) Application Programming in ANSAware – Document RM.102.02. APM, Poseidon House, Castle Park, Cambridge CM3 0RD, UK.

Barbacci M., Weinstock C., Doubleday D., Gardner M., Lichota R. (1993) Durra: a Structure Description Language for Developing Distributed Applications, *IEE Software Eng. Journal*, **8:2**, 83–94.

Becker K., Raabe U., Sloman M., Twidle K. (eds.) (1993) Domain and Policy Service Specification. IDSM Deliverable D6, SysMan Deliverable MA2V2. Available by FTP from dse.doc.ic.ac.uk.

Crane S., Twidle, K. (1994) Constructing Distributed UNIX Utilities in Regis. In *Proc. Second Int. Workshop on Configurable Distributed Systems*, IEEE Computer Society Press, 183–189.

Harter A., Hopper A. (1994) A Distributed Location System for the Active Office, *IEEE Network*, Jan./Feb. 1994, 62–70.

Kramer J., Magee J. (1990) The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Software Eng.*, **SE-16:11**, 1293–1306.

Kramer J., Magee J., Sloman M., Dulay N. (1992) Configuring Object-based distributed programs in REX, *IEE Software Eng. Journal*, **7:2**, 139–140.

Kramer J., Magee J., Ng K., Sloman M. (1993) The System Architect's Assistant for Design and Construction of Distributed Systems. In *Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, 284-290.

Leser N. (1993) The Distributed Computing Environment Naming Architecture. In IEE/IOP/BCS *Distributed Systems Engineering,* **1:1**, 19–28.

Magee J., Dulay N., Kramer J. (1994) REGIS: A Constructive Development Environment for Distributed Programs. In IOP/IEE/BCS *Distributed Systems Engineering,* **1:5**, 304–312.

Magee J. (1994) Configuration of Distributed Systems, Chapter 18 of *Network and Distributed Systems Management* (ed. Sloman M.), Addison Wesley, 483–497.

Moffett J., Sloman M. (1993) User and Mechanism Views of Distributed System Management. IEE/IOP/BCS *Distributed Systems Engineering*, **1:1**, 37–47.

Moffett J. (1994) Specification of Management Policy and Discretionary Access Control. Chapter 17 of *Network and Distributed Systems Management* (ed. Sloman M.), Addison Wesley, 455–480.

Sloman M., Moffett J. (1989) Domain Management for Distributed Systems. *Integrated Network Management I*, (eds. Meandzija B., Westcott J.), North Holland, 505–516.

Sloman M., Magee J., Twidle K., Kramer J. (1993) An Architecture for Managing Distributed Systems. In *Proc. 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, 40–46.

Sloman M., Twidle K. (1994) Domains: A Framework for Structuring Management Policy. Chapter 16 of *Network and Distributed Systems Management* (ed. Sloman M.), Addison Wesley, 433–453.

Strutt C. (1991) Dealing with Scale in an Enterprise Management Director. *Integrated Network Management II* (eds. Krishnan I., Zimmer W.), North Holland, 577–593.

Twidle K. (1993) Domain Services for Distributed Systems Management, PhD Thesis, Department of Computing, Imperial College.

Zimmermann M., Drobnik O. (1994) Specification and Implementation of Reconfigurable Distributed Applications. In *Proc. Second Int. Workshop on Configurable Distributed Systems*, IEEE Computer Society Press, 23–35.