A PRODUCT LINE ARCHITECTURE FOR A NETWORK PRODUCT

D E Perry

Rapporteur: Dr Robert Stroud

A Product Line Architecture for a Network Product

Dewayne E. Perry
Bell Laboratories
600 Mountain Ave
Murray Hill, NJ 07974 USA
+1.908.582.2529
dep@research.bell-labs.com

ABSTRACT
Given a set of related (and existing) network products, the goal of this architectural exercise was to define a generic architecture that was sufficient to encompass existing and future products in such a way as to satisfy the following two requirements: 1) represent the range of products from single board, centralized systems to multiple board, distributed systems; and 2) support dynamic reconfigurability.

We first describe the basic system abstractions and the typical organization for these kinds of products. We then describe an instance of the resulting architecture and show how these two requirements have been met. Our approach combines the two requirements neatly into an interdependent solution - though one could easily separate them into independent ones.

We use a late binding approach in such a way that it solves both requirements. The three important architectural components that provide this are the system model and data, the reconfiguration manager and the command broker. The system model and data enables the system to evolve itself dynamically via reflection; the reconfiguration manager uses the system model as the basis for dynamically reconfiguring the system; and the command broker is an object request broker mechanism that provides the necessary indirection and infrastructure to provide location transparency.

I then address the ubiquitous problem of how to deal with the problem of multiple dimensions of organization. In any type of system there are several competing ways in which the system might be organized. The issue arises of how to address the other means of organization once the primary dimension has been chosen. I show how architectural styles can be an effective mechanism for dealing with such issues as initialization and exception handling in a uniform way across the system components.

Keywords
Software Architecture Case Study, Dynamic Reconfiguration, Distribution-Free Architecture, Architecture Styles, Multiple Dimensions of Organization

1 Introduction

This study represents a snapshot in the process of constructing a generic architecture for a product line of network communications equipment — it is not *the* architecture for the product line. The purpose of this paper is to present several critical issues relevant to the architecture for the product line, to discuss the implications of those issues, and to describe several intersting architectural techniques that solve these issues in interesting ways. We provide enough of the domain specific architecture to give the appropriate context for the part of the architecture we focus on.

We first provide the context for the study (the product line domain, the current and desired states of the product line, and a basic view of the products). We then explore the implications of the primary requirements and what is needed at the architectural level to satisfy those requirements. On this basis, we the describe our solution and the motivation behind our choices. Finally, we summarize what we have done and lessons we learned in the process.

2 Context

The product line consists of network communication products that are hardware event-driven, real-time embedded systems. They have high reliability and integrity constraints and as such must be fault-tolerant and fault-recoverable. Since they must operate in a variety of environments, they are "hardened" as well.

The current state of the products in this product line is that each product is custom built to a customer's specifications with hard-wired hardware and specially build software. To evolve one of these products, one must specify a new instance and have it specifically built.

The basic abstraction for these products is that of a *con-*

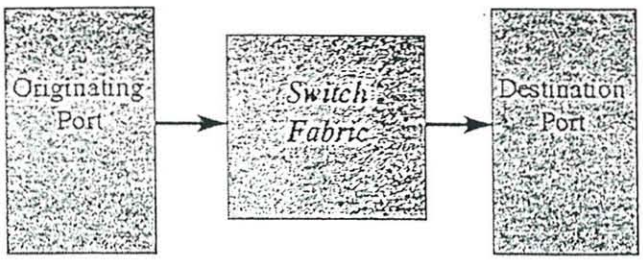


Figure 1: **Basic Abstraction: Connection.** A connection consists of an originating port connected via a switch fabric to a destination port.

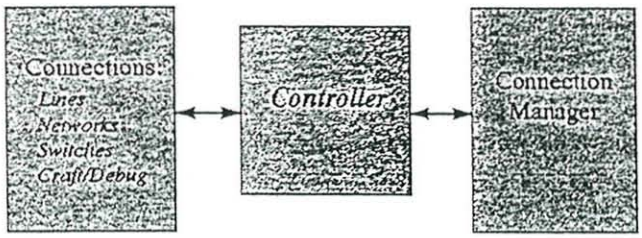


Figure 2: **Basic Hardware/Software System:** consists of three logical elements: connections, controllers and a control manager.

nection. A connection consists of an originating port connected through a connection or switch fabric to a destination port. The connections range from static ones (which once made remain in existence until the devices attached to the ports are removed) to dynamic ones (which range from simple to very complex connections that vary in the duration of their existence) — see Figure 1.

The typical system structure for these products (see Figure2) consists of a set of connections such as communication lines, switches, other network connections, and craft and debugging interfaces. These devices have various appropriate controllers that are handled by a connection manager which establishes and removes connections according to hardware control events.

Figure 3 shows a typical architecture for such network communication products layered into service, network and equipment layers. Within each layer are the appropriate components for the functionality relevant to that layer.

3 Basic Requirements

The basic requirements for the product line architecture we seek are:

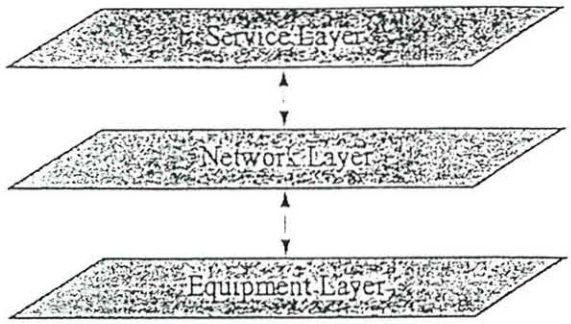


Figure 3: **Typical Domain-Specific Architecture:** a structure of three layers consistent with the standard network model.

- To cover the large set of diverse product instances that currently exist and that may be desired in the future
- To support dynamic reconfiguration so that the products existing in the field can evolve as demands change for new and different kinds of communication.

Thus the desired state of the product line is that products can be reconfigured as needed with as little disruption as possible (but not requiring continuous service). For the hardware, this entails common interfaces for the various communication devices and plug compatible components. For the software, this entails a generic architecture for the complete set of products and and software support for dynamic reconfiguration of the system.

The first question then is how do we create a generic architecture that covers the entire range of products in the product line? These products range from simple single board systems to complex, multi-board and distributed systems. If we address the issue of distribution at the architectural level, then that implies that distribution is a characteristic of all the instances. What then do we do with simpler systems? A separate architecture then defeats the goal of a single generic architecture for the entire product line.

One answer to this question is to create a *distribution free* architecture [3] and thus bury distribution down into the design and implementation layers of the system construction process. In this way, distribution is not an architectural issue as such.

However, this decision does have significant implications at the architectural level about how the issues of distribution are to be solved. First, the system needs a model

of itself that can be used by the appropriate components that must deal with issues of distribution. For example, the component handling system commands and requests must know where the components are in order to schedule and invoke them. Thus, second, we need a command broker that provides location transparent communication, that is configurable, that is priority based and that is small and fast. The last two requirements are due to the real-time requirements on the system as a whole. Finally, the components need to be location independent in order to be useful across the entire range of products.

To satisfy the requirement for dynamic reconfiguration, it is necessary only to minimize down time. We do not need to provide continuous service. However, we need to be able to reconfigure the system *in situ* in any number of ways from merely replacing line cards to adding significantly to the size and complexity of a system (for example, changing a simple system into a complex distributed one) in the hardware and from changing connection types to adding and deleting services in the software.

As with the issue of distribution, reconfigurability requires a model of the system and its resources, and obviously, a reconfiguration manager that directs the entire reconfiguration process both systematically and reliably. For this to work properly, the components have to have certain properties akin to location independence for a distribution-free system. In this case, we need configurable components. We shall see below that these necessary properties can be concisely describe in an architectural style [1].

To ensure that any reconfiguration results in an a complete and executable state, consistency and completely analysis must be done to ensure that the resulting system is not missing anything and that all the pieces work together properly. The question arises then as to where this part of reconfiguration manager should be. Given the space and economic considerations of the systems, we chose to have the consistency checking done outside the bounds of the system architecture.

4 Architectural Solution

By and large, a product line architecture is the result of pulling together various existing systems into a coherent set of products. It is essentially a legacy endeavor: begin with existing systems and generalize into a product line. There are of course exceptions, but in this case the products preceded the product line.

The appropriate place to start considering the generic architecture is to look at what had been done before. In this case we draw on the experience of two teams for two different products and use their experience to guide us in our decisions.

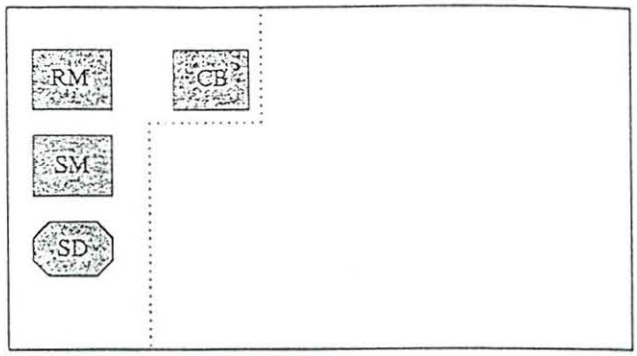


Figure 4: Reconfiguration Components: System Model (SM), System Data (SD), Reconfiguration Manager (RM), and Command Broker (CB).

As in any system of any complexity, there are multiple ways of organizing [2] both the functionality and the various ways of supporting nonfunctional properties. In this case, we see two more or less orthogonal dimensions of organization: system objects and system functionality. System objects reflect the basic hardware orientation of these systems: packs, slots, protection groups, cables, lines, switches, systems, etc. System functionalities reflect the things that the systems does: configuration, connection, fault handling, protection, synchronization, initialization, recovery, etc.

Given the two dimensions, the strategy in the two developments was to organize along one dimension and distribute the other throughout that dimension's components. In the one case, they chose the system object dimension, in the other they chose the system functionality dimension. Both groups felt their solutions were unsatisfactory and were going to choose the other dimension on their next development.

Our strategy then was to take a hybrid approach: choose the components that are considered to be central at the architectural level and then distribute the other aspects throughout those components — a mix and match approach. The question then is how to gain consistency for the architectural considerations that get distributed over the architectural components. We illustrate the use of architectural styles as a solution to this problem in two interesting cases below.

For the satisfaction of the product line requirements we have the four components illustrated in Figure 4 the command broker (CB), the reconfiguration manager (RM), the system model (SM) and the system provisioning data (SD).

The system model and system data provide a logical model of the system, the logical to physical mapping o

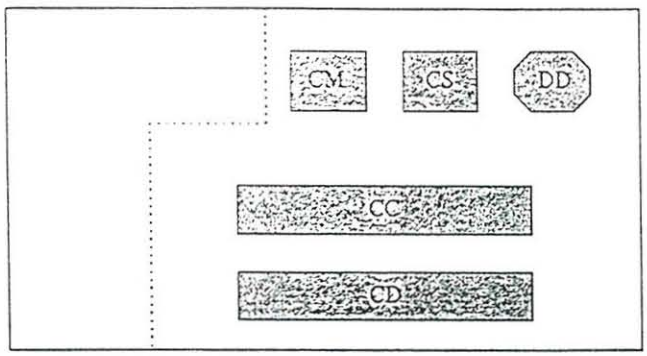


Figure 5: **Domain-Specific Components:** Connection Manager (CM), Connection Services (CS), Dynamic Data (DD, Connection Controller (CC), and Connection Devices (CD).

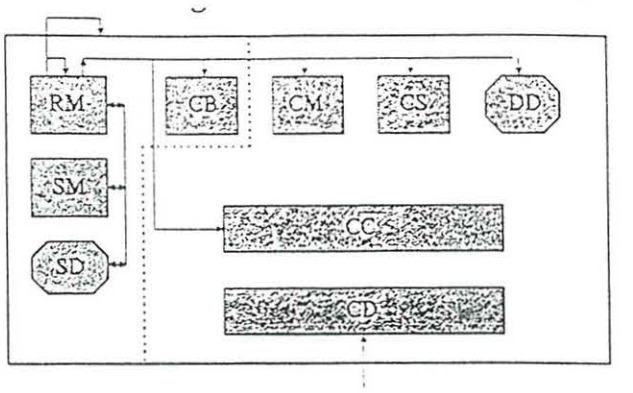


Figure 6: **Reconfiguration Connections.** The reconfiguration manager is connected in various ways to all the components in the system, including itself and the system as a whole.

the various elements in the system configuration, and priority and timing constraints that have to be met in the scheduling and execution of system functions.

The command broker uses the system model to drive its operation scheduling and invocation. System commands are made in terms of logical entities and the logical to physical mapping is what determines where the appropriate component is and how to schedule it and communicate with it.

Reconfiguration is split into two parts: reconfiguration generation and reconfiguration management. the reconfiguration generator is outside the architecture of the system and ensures that the reconfiguration constraints for completeness and consistency of a configuration are satisfied. It also ensures that the configured system is minimal [?], a requirement due to both space and time limitations.

The reconfiguration manager directs the termination of components to be removed or replaced, performs the component deletion, addition or replacement, does the appropriate registration and mapping in the system model, and handles startup and reinitialization of new and existing components. Special care has to be taken in the construction of the reconfiguration manager so that it can properly manage self-replacement, just as special care has to be taken in any major restructuring of the hardware and software.

For the domain-specific part of the architecture we have chosen as the basic architectural elements, as shown in Figure 5, the connection manager (CM), the integrity manager (IM), the connection services component (CS), the connection controllers (CC), and the connection devices (CD). These components represent our choices for the architectural abstractions of both the critical objects and the critical functionality necessary for our product line. Of these, the integrity manager is a logical component whose functionality is distributed throughout the other components.

The reconfiguration interactions shown in Figure 6 illustrate how the reconfiguration manager is intimately tied to both the system model and the system provisioning data. This part of the reconfiguration has be handled with care in the right order to result in a consistent system. Further, the reconfiguration manager interacts with itself and the entire configuration as well as the individual components of the system: terminate first, preserving data, reconfigure the model and provisioning, and then reconfigure the components. There are integrity constraints on all of these interactions and connections.

The reconfigurable component architectural style that must be adherred to by all the reconfigurable components has the following constraints:

- The component must be location independent
- Initialization must provide facilities for start and restart, rebuilding dynamic data, allocating resources, and initializing the component
- Finalization must provide facilities for preserving dynamic data, releasing resources, and terminating the component

While we have not used the typical network model as the primary organizing principle for the architecture, it does come into play in defining the hierarchy or decomposition of several of the basic domain specific system components: the connection manager, the connection services, and the connection controller.

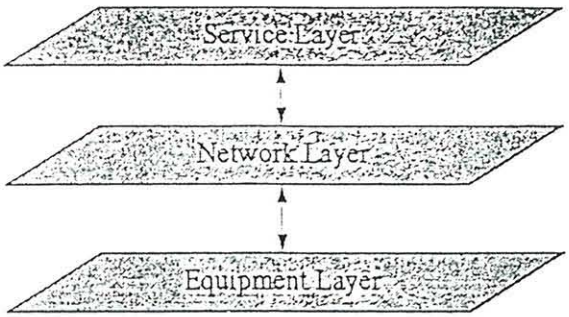


Figure 7: **Domain Specific Component Decomposition.** The traditional layering forms the basis of the subarchitectures several of the basic domain specific components.

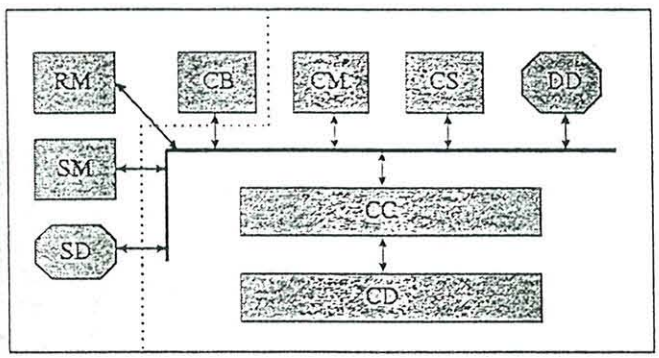


Figure 8: **Architectural Connections.** A software bus provides the primary control and data connectors among the system components.

A software bus provides the primary connector amongst the system components for both control and data access. There are other connectors as well, but they have not been necessary for the exposition of the critical aspects of the generic architecture. There are both performance and reliability constraints that must be met by this primary connector. The manager of the bus is the command broker.

We had mentioned earlier that the integrity manager was a logical component that was distributed across all the architectural components. As such there is an integrity connector that hooks all the integrity management components together in handling exceptions and recovering from faults. We had also indicated that the part of the integrity management would be defined as an architectural style that all the system components had to adher to. This style is defined as follows:

- Recover when possible, otherwise reconfigure around the fault
- Isolate a fault without impacting other components
- Avoid false dispatches
- Provide mechanisms for inhibiting any action
- Do not leave working components unavailable
- Enable working in the presence of faults
- Recover from single faults
- Protect agains rolling recoveries
- Collect and log appropriate information
- Map exceptions to faults
- Enable sequencing of recovery actions

5 Summary and Lessons

We have explored several interesting techniques to achieve a generic architecture that satisfied both the domain-specific requirements and the product-line architecture requirements.

To achieve an appropriate domain-specific architecture, we chose a hybrid approach in which we selected what we considered to be the critical elements from two orthogonal dimensions of organization. We then defined architectural styles to ensure the consistency of the secondary components distributed throughout the primary components. We defined a software bus as a general connector among the components subject to both performance and reliability constraints. This latter is especially important where the underlying implementation and organization is distributed across several independent physical components.

To achieve the appropriate goals of the product line generic architecture and enable dynamic reconfiguration, we chose a data-driven, late binding and reflective approach. This enabled us to solve both the problem of centralized and distributed systems and the problem of reconfiguration with essentially the same mechanisms.

As to lessons learned:

- To quote an old saying "there are many ways to skin a cat". So too there are many ways to organize an architecture, even a domain specific one. Because there are multiple possible dimensions of organization, some orthogonal, some interdependent, experience is a critical factor in the selection of critical architectural elements, even when considering only functional, much less when considering non-functional, properties.

- It is extremely important for any architecture, design or implementation to have appropriate and relevant abstractions to help in the organizing of a system. An example in this study is that of a connection as the central abstraction. Concentration on the concepts and abstractions from the problem domain rather than the solution domain is helpful in this respect.

- Properties such distribution-independence or platform-independence are extremely useful in creating a generic product line architecture. They do, however, come at a cost in terms of requiring appropriate architectural components that enable those particular properties.

- Architectural styles are an extremely useful mechanism in ensuring uniform properties across architectural elements, especially for such considerations as initialization, exception handling and fault recovery where local knowledge is critical and separated by various kinds of logical and physical boundaries.

Acknowledgements

Nancy Lee was my liaison with the architectural group on this project. She helped in many ways, not the least of which was making project data and documents available for me to write up this case study. The system architects on the project as a whole were very tolerant of an outsider working with them. However, we achieved a good working relationship combining their domain expertize with my research investigations and together with a willingness to explore alternative

possibilities.

REFERENCES

[1] Dewayne E. Perry and Alexander L Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17:4 (October 1992)

[2] Dewayne E. Perry. Shared Dependencies. In *Proceedings of the 6th Software Configuration Management Workshop*, Berlin, Germany, March 1996.

[3] Dewayne E. Perry. Maintaining Minimal Consistent Configurations. Position paper for the 7th Software Configuration Management Workshop, Boston Massachusetts, May 1997.

[4] Dewayne E. Perry. Generic Architecture Descriptions. In *ARES II Product Line Architecture Workshop Proceedings*, Los Palmas, Spain, February 1998.

A Case Study in Product Line Architectures

Dewayne E. Perry

Bell Laboratories
Room 2A-429
600 Mountain Ave
Murray Hill NJ 07974

dep@research.bell-labs.com
www.bell-labs.com/~dep/

Context

- A snapshot during the architectural process for this product line (ie, not THE final product line architecture)
- Basic requirements
  - Cover a large class of diverse instances in the same application domain
  - Support dynamic reconfiguration
- Simplification of non-relevant issues

Outline

- ***Background and context***
- Satisfying the basic requirements
- Our architectural solution
- Summary

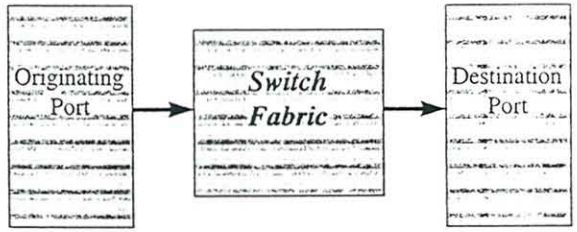
Product Line Domain

- Network Communication Product
- Real time, embedded system
- HW event driven
- High reliability, high integrity
- Fault-tolerant, fault-recoverable
- Hardened - to operate in a variety of environments

Current State of Product Line

- ✳Custom built to customer specification
- ✳Hard-wired hardware
- ✳Hard/hand-coded software
- ✳To evolve: build new hardware and software

Basic Abstraction: *Connection*



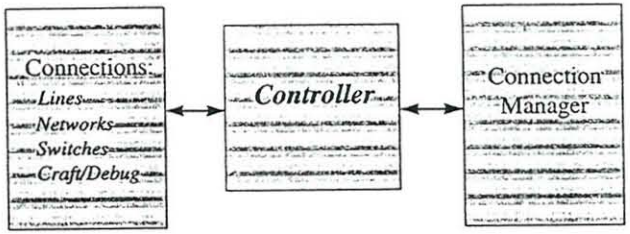
Target State of the Product Line

- ✳Dynamic reconfiguration of both HW and SW
- ✳Hardware
  - common interfaces
  - plug compatible components
- ✳Software
  - generic architecture
  - common platform
  - plug and play

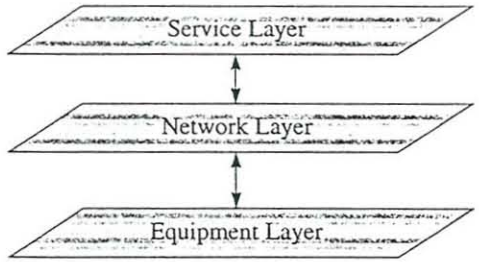
Basic Abstraction: Connections

- ✳Variety of connections from
  - relatively static to
  - dynamic, simple to complex
- ✳Variety of connection machines from
  - simple one board, centralized systems to
  - multiple board, distributed systems

Basic HW/SW System



Typical Architecture



Basic System

- Devices of various sorts that are used for connections to various kinds of network components
- Controllers for those devices
- A connection manage to establish and remove connections

Outline

- Background and context
- *Satisfying the basic requirements*
- Our architectural solution
- Summary

Whither Dynamic Reconfiguration

- ✳Do not need continuous availability
- ✳Do need to minimize downtime
- ✳Ability to change in situ
  - overall organization: centralized to distributed
  - change connections
  - add, replace, delete services

Whither Distribution

- ✳Part of architecture?
  - then all instances must be distributed
  - but some are single processor systems
- ✳Distribution Independence
  - emphasis on components and interactions
  - bury distribution in supporting platform

Implications of Reconfigurability

- ✳Model of system and resources
- ✳Configuration Manager
- ✳Configurable component style
- ✳loci of reconfigured system
  - generation
  - analysis
  - linking

Implications of Distribution Free

- ✳Need an object request broker
  - location transparent communication
  - configurable
  - priority-based
  - small and fast
- ✳Location independent components
- ✳Model of the system

Outline

- ✳Background and context
- ✳Satisfying the basic requirements
- ✳*Our architectural solution*
- ✳Summary

Two Possible Dimensions

- ✳System Objects:
  - pack, slot, protection group, cable, line, switch, system
- ✳System Functionality:
  - configuration, connection, fault, protection, synchronization, initialization, recovery

Initial Considerations

- ✳What are the possibilities?
- ✳What was past experience?
- ✳Initial strategies

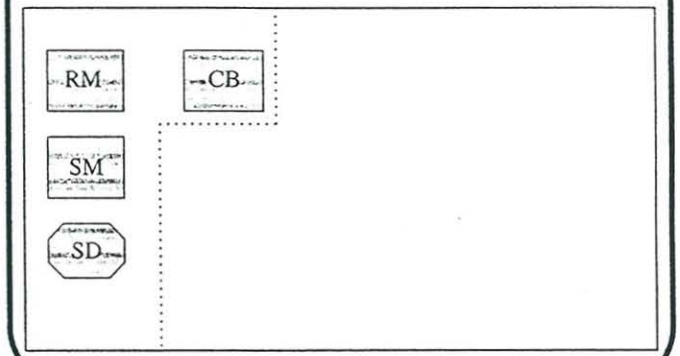
Experience

- ✳Organize on one dimension, distribute the other
- ✳Previous product architecture experience
  - one group: system objects
  - another: system functionality
- ✳Evaluation of both groups
  - neither solution satisfactory
  - going to do the other dimension

Initial Strategy

- ✳Choose some components in each dimension as the primary architectural components
- ✳Define the distributed components as SW Architectural Styles
  - e.g., constraints on initialization
    - common across all components
    - consistent across all components
  - e.g., fault detection, recovery, etc..

Distr/Reconfig Components



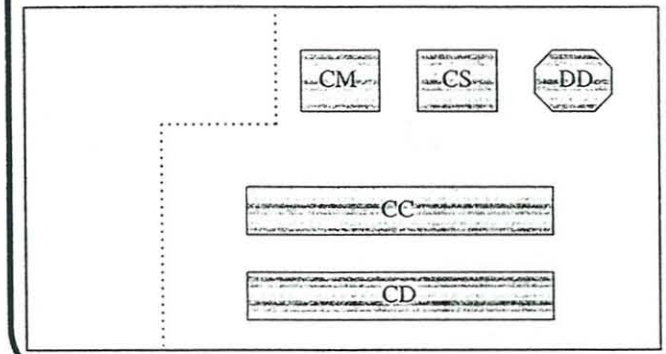
Distr/Reconfig Components

- ✳CB - Command Broker
- ✳SM - System Model
- ✳SD - System Data
- ✳RM - Reconfigure Manager

Domain-Specific Components

- ✳CM - Connection Manager
- ✳IM - Integrity Manager
- ✳CS - Connection Services
- ✳CC - Connection Controllers
- ✳CD - Connection Devices (HW)

Domain-Specific Components



Reconfiguration Components

* Reconfiguration Generation (RG)
  - Outside the fielded system
  - Component generation
  - Completeness/consistency analysis
  - Configuration minimality

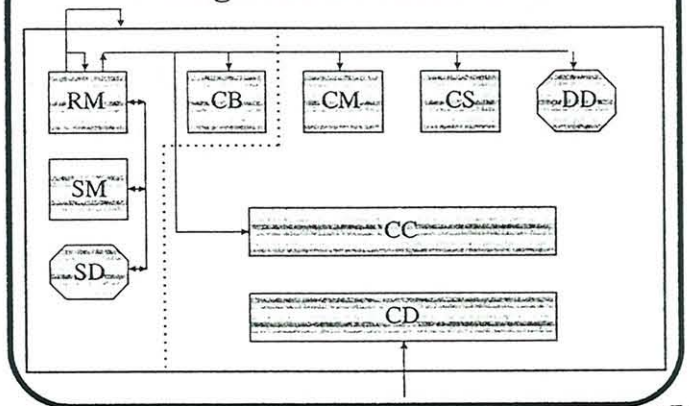
Distribution Components

* System Model/Data (SM/SD)
  - Logical Model
  - Logical to Physical Mapping
  - Priority/Timing constraints
* Command Broker (CB)
  - Operation invocation
  - Operation scheduling

Reconfiguration Components

* Reconfiguration Manager (RM)
  - Termination of components
  - SM, SD, Component update
  - Registration/Linking
  - Initialization
  - Reflection to be able to replace self

Configuration Connections



Style for Reconfigurable Comp's

- Location independent
- Initialize:
  - start/restart, rebuild dynamic data, allocate resources, initialize operation
- Finalize:
  - preserve dynamic data, release resources, terminate operation

Reconfiguration Connections

- RM to self - in case of RM replacement
- RM to entire configuration
- RM to individual components
  - termination first, preserve data
  - reconfigure model and provisioning
  - reconfigure components
- Integrity constraints on connections

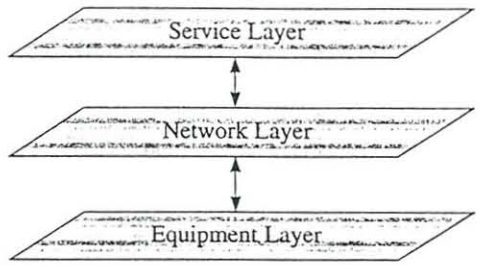
Reconfiguration Generation (RG)

- Problem: maintaining a minimum configuration in the access/transport boxes
  - typically limited space
  - avoid clutter of unused software components
  - minimize reconfiguration time and expense

Minimal Reconfiguration Solution

- AED is the set of architectural elements and their dependencies
- CC is the current architectural element configuration
- D(X) is the transitive closure of X in AED
- ADD(AE) = D(AE) - D(CC)
- DELETE(AE) = D(AE) - D(CC - AE)
- Do ADDs first

DS Component Decomposition



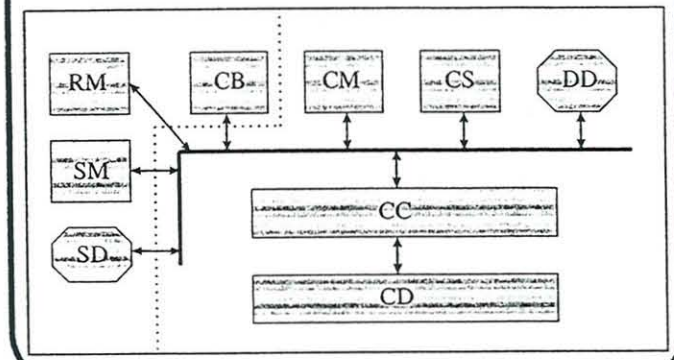
DS Architectural Structure

- CM/CS - use typical architecture for decomposition/layering
  - service layer
  - network layer
  - equipment layer
- IM/CC - distribute using styles

DS Architectural Connections

- Software bus for
  - Control of interactions
  - access to dynamic and system data
- Performance constraints
- Reliability constraints

Architectural Connections



IM Exception Handling Style

* Recover from single faults
* Protect against rolling recoveries
* collect, log appropriate information
* map exceptions to faults
* enable sequencing of recovery actions

IM Exception Handling Style

* Recover when can, else reconfigure around fault
* Isolate fault without impacting other components
* Avoid false dispatches
* Provide mechanisms for inhibiting any action
* Do not leave working components unavailable
* Enable working in the presents of faults

Outline

* Background and context
* Satisfying the basic requirements
* Our architectural solution
* *Summary*

Summary

*Techniques for distribution-free and dynamically reconfigurable architecture
- Data-driven
- Late dynamic binding
- Reflection

Summary

*Techniques for Domain Specific Organization
- Primary components - architectural elements
- secondary components - architectural styles
- classes of interactions
  - different connectors
  - with different constraints

DISCUSSION

Rapporteur: Dr Robert Stroud

Lecture Two

Professor Brooks asked why a small ORB had been used. Dr Perry explained that this had been for economic reasons to keep things tight and inexpensive to produce.

Mr Hutt asked if there were any numbers for the reconfiguration requirements. No - but some numbers had been emerging just as the project was cancelled. Professor Randell expressed surprise that the abstract details of the design hadn't been driven by some sort of quantitative data from the field, even just ball park figures or ratios. Dr Perry replied that perhaps the numbers had been implicit and simply hadn't entered into discussions about the basic organisation of the system.

Professor Henderson wanted to know how they knew that off the shelf components would not be good enough. Because they'd tried them, Dr Perry replied. Professor Henderson replied that this sounded like an academic solution but Dr Perry said definitely not. He had been working with real architects and had built a lot of large systems himself in any case so he'd certainly paid his implementation dues.

Mr Hutt said that the whole reason for chasing metrics was because they drove decisions and influenced the cost. Dr Perry replied that cost had certainly been an issue and had led to the requirement for a minimum configuration. The system architects, who knew the requirements and had built several such systems before, had not objected to the proposed structure.

Professor Shaw argued that there were three possibilities: they knew the numbers but couldn't publish them, the numbers were known implicitly but the discussions had been qualitative, or the numbers had not been thought of. Dr Perry disagreed but said that the first two possibilities certainly didn't apply.

Professor Randell thought that perhaps there had been no need to write down requirements that were obvious to everybody and Mr Jackson argued that the requirement for metrics everywhere was somewhat exaggerated. There was no need to study options that obviously weren't sensible such as a cheap method of connection that involved a loss of service for six hours.

Professor Shaw asked about the cost of acquiring information and Dr Perry replied that the cost always comes out of something else.

Professor Turski asked whether Dr Perry was trying to do something analogous to transporting a whale in a car but was told that this was definitely not the case. Because the family of systems being designed covered a wide span, there needed to be a wide span in performance and several versions of each component were required in order to achieve this.

Referring to a function box in one of the architecture diagrams, Professor Randell asked in what sense faults could be considered to be part of the functionality of the system. Dr Perry explained that this part of the system was concerned with fault handling but Mr Jackson joked that faults were a main part of the functionality of many systems!

Mr Jackson asked why the Command Broker only appeared in one part of the system. It wasn't clear that this was an architectural issue - couldn't the Command Broker just be hardwired as a jump table? Apparently it was needed for distributed recovery.

Professor Brooks asked how the teams of people working on the project had been organised. There had been two distinct groups with experience of building such systems in two different ways - had they deliberately been mixed up? No - each group was fairly small and tightly knit and the groups had been kept together.

Professor Shaw asked about the difference between the two kinds of arrow (single and double headed) used in the diagrams. A double headed arrow between two components meant that there was interaction in both directions - the same line was used to mean both communication and reconfiguration. Conversely, a single headed arrow meant that one component was responsible for replacing another and didn't represent an interaction.

Professor Shaw also asked whether configuration meant changing parameters or replacing components. It could apparently mean both.

Another member of the audience asked about the meaning of the CD component which was completely disconnected from the rest of the system. This was because devices got changed from outside the system although it was still necessary to change the appropriate device controllers.

Mr Jackson asked what was meant by preserving dynamic data. This was something that had to be done across reconfiguration to preserve the state of the system. Although the semantics would remain the same, rebuilding the state might involve some transformation and this was done by each component as appropriate using local knowledge. Professor Shaw asked about changes in global data - these were dealt with by the reconfiguration data.

Professor Shaw observed that Dr Perry had made an interesting shift between two successive diagrams. The first diagram showed the information relationship whilst the second diagram made visible the piece of the infrastructure that carried the information. This was the topological equivalent of a software bus. But all the connections were mediated by the Command Broker. The relationship between parts of the system had become hidden - she didn't like using the Command Broker as a connector because it buried important detail. Dr Perry argued that it wasn't being used in this way because all the interactions had been combined.

Mr Hutt thought that the diagram missed out layering issues, in particular the intercomponent communication layer. Dr Perry disagreed, saying that the diagram was an extreme simplification and in fact components interacted at each level. There was a basic problem trying to get everything into one diagram and provide a reasonable overview.

Professor Johnson wanted to discuss what the best view of the system was. He argued that there were two possible views - reconfiguration and execution - but he claimed that execution was the wrong model. Instead of making a connector into an object. wasn't it better to represent it as an arrow? The internal structure didn't affect how the system worked and it was only important to understand how the components interacted.

Professor Shaw asked whether the primary objective of the diagrams was to explain the functionality of the systems or to show their family character. She felt that which slide was most appropriate depended on the answer to this question. The first slide explained the interactions during reconfiguration and how the reconfiguration manager related to the rest of the system. Thus, it was explaining the commonality between the systems which was better for describing the product family.

Professor Randell asked how consistency could be ensured when the same thing was presented from several different viewpoints. For example, the viewpoints might involve the same objects but show different relationships between them. But Mr Jackson pointed out that this didn't apply in this case - the diagrams had different objects in them such as additional connectors.

Professor Johnson argued that distribution was part of the solution domain and that architectural designs should concentrate on the problem domain. There had to be a balance between the two but he always taught his students to concentrate on the problem domain because he knew the solution domain would sneak in anyway! However, Dr Perry believed that it had been essential to start from the basic abstraction of a "connection machine" (i.e. a machine for managing connections) and that the final outcome would not have been as good otherwise.

Another member of the audience complained about the slide in which all the boxes were connected to each other in a CORBA-like fashion. This was just the underlying technology used to implement the architecture - why should it form part of the diagram? Nobody ever put the programming language (e.g. C++ or Smalltalk) into such diagrams. It should be implicit that you needed a mechanism for communication, just as it was implicit that you needed a programming language. Dr Perry agreed - he didn't like including this detail either but it was a matter of "truth in advertising".

Professor Balzer said that the question could be asked in a different way - was there a software bus and if so, why did you choose to draw it? However, Mr Jackson observed that there was some value in the diagram because it showed that the CC/CD connection did not go through the bus. Dr Perry agreed and said that the diagram conflated several different relationships.