A later version of this paper will appear in the Int Journal of Software Engineering and Knowledge Engineering, sometime in 2001.

Software Resource Architecture

C. M. Woodside,
Dept. of Systems and Computer Engineering, Carleton University,
Ottawa, Canada K1S 5B6
email: cmw@sce.carleton.ca
Dec 7, 2000

Abstract

Performance is determined by a system's resources and its workload. Some of the resources are software resources which are embedded in the software architecture; some of them are even created by the software behaviour. This paper describes software resources and resource architecture, and shows how resource architecture can be determined from software architecture and behaviour. It considers how resource architecture emerges during design, the relationship of software and hardware resources, some classes of resource architecture, and what they can tell us about system performance. Other uses of resource architecture are, to analyze deadlocks, to understand special software architectures developed for demanding situations, and to analyze how subsystems fit together when they share resources. Resource architecture can be described using description languages (ADLs) developed for software architecture.

1 Introduction

Software is just a set of instructions that govern the use of resources such as processors, memory, buses, and peripheral devices. The software also introduces artificial or logical resources into the system to coordinate the use of physical resources such as memory or peripheral devices, (for example, a semaphore), to protect the integrity of data (for example, a lock), and for many other purposes.

Together the software and hardware resources govern the performance of a software system, in the sense of its response delays or its capacity to handle traffic. They can prevent a program from proceeding, until some resource can be allocated to the program. This aspect of *authority to proceed* is common to both software resources and hardware resources; we wish to emphasize their similarities and common properties, in an overall resource architecture for a system.

Resources are seldom treated systematically and carefully in software design, except perhaps for processor time. This may be intentional, as in a design which is intended to be useable over a wide variety of physical platforms (so it deliberately avoids consideration of physical resources). More often resource usage is an emergent property of the design, and resource interactions may introduce undesirable delays. In some systems the patterns of resource use have enough structure that we can identify a resource architecture, while in other cases the patterns are chaotic and the architecture is not obvious.

Software architecture is essentially a system description in terms of *components* and their *interactions*. Components may be clients and servers, databases, filters, layers, modules or subsystems. Interactions include messages, calls, communications protocols, database access protocols, and multicasts. An interaction is attached at both ends to distinct *ports*, which are associated with defined roles for the partners in the interaction.

The value of architecture is that it creates a coherent overall structure which guides, contains and maintains the functional details. More discussion of software architecture is given by Shaw and Garlan [24], Bass et al. [1] , and Hofmeister et al. [9].

Coherent overall patterns of resource use are similarly important, particularly for efficient operation. Some software components such as storage (buffers, files) are essentially just resources, while others, such as (e.g. operating systems processes, blackboards, databases) have resource attributes along with other roles. Poor resource use results in resource holding times that are longer than necessary, in fragmented operations which acquire and release resources many times, and in logical bottlenecks, which are discussed below.

Resource aspects of software are mentioned with regard to concurrency decisions, for instance in [24], and in architecture evaluation, for instance in [1], [4]. Klein, Kazman and Clements specifically considered performance as an example of attribute-based architecture evaluation, in [12]. Shaw has recently addressed the impact of changing resource capabilities in “open resource coalitions” which may assemble resources dynamically as they run [25]. However there has been little systematic consideration of resource attributes of software architecture. What we find are statements along the lines that a certain alternative software architecture is judged to be good for performance, because it permits concurrency (see e.g. [4]).

If we understand resource architecture better, we may be able to:

- develop resource-dominated software architectures from scratch, for applications in which quality of service is important,
- develop a performance model for a given software architecture, and make performance predictions,
- plan deployment of a given software architecture in different versions with different resources, for example over varying scales.

Performance approaches such as layered queueing models have been developed to study these issues, particularly the latter two. A key notion in layered queueing is a resource entity that can take the role of a server, in processing a request, and then turn around and act as a client in requesting service from some lower level resource (e.g. [28], [32], [21]). This is natural behaviour for a software server, and leads to a model with abstract entities that correspond to software resources. Layered queueing has been applied to web servers [5], transaction processing [10], data routers [18] and distributed databases [26].

Besides performance-related properties, resource architecture also identifies possibilities of resource deadlock due to resource request patterns.

This paper investigates the connection between a software architecture and its resources, in order to promote all three of the goals listed above, and others which may follow from a clearer understanding. The idea and the essential features of a resource architecture are described, including some common styles (some of which resemble styles of software architecture). We will see that the resource architecture may be strongly structured, or not, and that when it emerges from a software architecture, the resources may or may not have a simple relationship to each other. The concept of resource architecture was described in an early version of this work [34]. The present paper adds detail, an architectural notation, and an approach to the systematic development of resource architecture models.

2 Resources in software systems

Consider an hypothetical image-processing system called HyperCam that executes the scenario shown in Figure 1, with operations to capture an image, process it

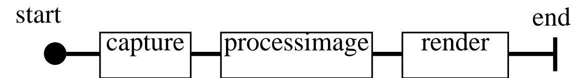


FIGURE 1. Scenario 1: a sequence of activities to be performed by the HyperCam system

according to the user’s instructions, and render and display the results. The scenario is shown as a Use Case Map [3], starting from the filled circle, following a path defined by the heavy line, and ending at the bar. The three high-level operations are shown by labelled boxes. A high-level architecture for the software is shown in Figure 2.

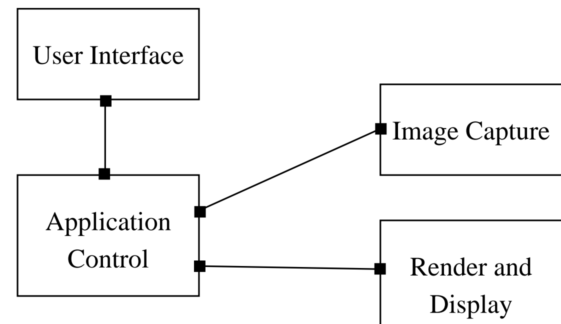


FIGURE 2. Conceptual software architecture for the HyperCam system

Now consider the resources that might be needed to execute the activities in Scenario 1. The term *resource* is often used to refer to three different kinds of entities:

1. *execution resources*, which host the execution of the operations and are usually system devices such as processors, interface devices, buses, and disk devices and controllers;
2. *logical resources* or *software resources*, which are essentially rights to proceed, and to execute certain operations which we shall call *resource-operations*. Examples of software resources include program

threads, semaphores, locks, buffers, access control permissions, and window flow control permissions.

3. *data resources*, being information which the program must access. The term Universal Resource Locator (URL) refers to this kind of resource, as does the Resource Description Framework (RDF) [15], which essentially refers to resources which are documents.

This work refers only to the first two types, because it concentrates on entities with access limitations, that must grant permission to proceed. Data resources as pure information are therefore not included, except the aspect of access permission to the data, which is regarded as a software resource.

Resource context

Figure 3 displays the use of resources for Scenario 1 as a shaded zones. At a point where the program must

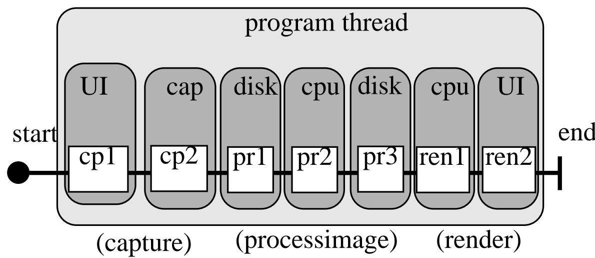


FIGURE 3. Scenario 1, with resources and resource contexts indicated for finer-grained activities

acquire a resource, the path enters a rounded shaded box representing the resource. The path stays in the box until it reaches a point where it gives up the resource. Each activity is thus surrounded by a set of shaded zones representing the resources that it requires.

The first resource shown is the program thread, which represents the operating system resources required to run a processor or thread. When the program is invoked the shell waits until these are assigned, and the program is started. The inner zones are the execution resources of the individual activities.

For simplicity of the diagram, the operations in the scenario have been broken down into small activities which are each executed by just one of the system devices. Thus "capture" in Figure 1 has been broken into "cp1" executed on the user interface device (perhaps a PC), and "cp2" executed on the digital image capture interface. The execution resources are the user interface device (UI), the image capture device (cap), the main image processor (cpu), and a disk subsystem, (disk). At a point where the execution path enters any shaded zone the program must request the resource and wait until it is granted. Each resource has a controller, scheduler or arbiter which makes the decision and manages a queue of waiting requests, and these controllers are implied in the figure, but not shown.

The set of resources surrounding each activity in Figure 3 is its *resource context.* The resource context of activity "cp1", for example, is the user interface UI and the program thread.

Using scenario analysis some of the resources for a program can be analyzed even before the software architecture is determined. Additional resources such as processes and threads will be created later in the software design.

Additional types of software resources are common. Figure 4 shows a modified version of Scenario 1 using a file server and a lock. The lock is an exclusive resource which is obtained before storing the image data as a file (and reading and writing additional files as processing proceeds). The program connot proceed until it has the lock. Similarly a file server thread is a resource which is required for file server operations.

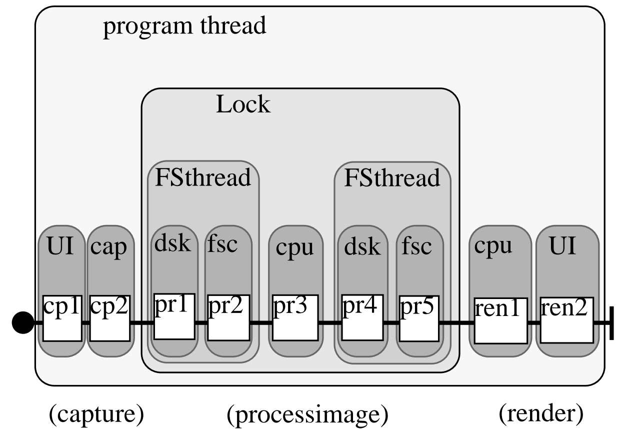


FIGURE 4. Scenario 2, with lock and file server resources, showing the resource context for each activity.

Figure 4 shows that resource contexts may be complex and may change rapidly. The Figure conceals some of this behaviour, as it is drawn in a rather abstract form. For instance the program may switch between the file server cpu (fsc) and the disk (dsk) many times.

Resource multiplicity

A resource identified in a context may be provided as a single resource, such that only one request can proceed at a time, or as a multiple resource, which is a pool of equivalent resources managed by a single controller. A multiple CPU could be provided by a symmetric multi-

processor; a counting semaphore allows multiple requests to pass up to its limit; a multi-threaded task allows multiple instances of the program to execute; a buffer pool can allocate all its elements separately. All these are examples of resource multiplicity.

A process with unlimited threads, such as a server which creates a thread per request, is an infinite set of resource tokens which actually cannot withhold permission to proceed. In this sense an unlimited resource pool is missing one of the key attributes of a resource, and removes the limitations associated with that resource in the architecture. On the other hand the removal may be illusory, for instance an unlimited thread pool is still limited by the availability of memory to instantiate the threads in.

Nested contexts

The resource contexts in the above figures are strictly nested, one within the other. Commonly, resource contexts are nested like this, however within a particular resource context there may be a variety of choices for further nested resources. This means essentially that resources are released in the reverse order to which they are obtained. One way to ensure this, when multiple program threads are in competition for a variety of resources, is to have a global ordering of resources and to always obtain multiple resources in this order. This strategy is used for instance with locks, to avoid locking deadlock, and it also ensures against deadlock with other resources. A weaker condition is to that a directed acyclic graph should exist, with resources as nodes, and all resource requests are made in the order given by the graph. This does not require a prior global ordering.

Strictly nested resource contexts lead to layered queueing models for performance, as described in Section 6. Non-nested resource contexts are certainly possible, as illustrated in Figure 5, However they are more difficult to represent and to understand than strictly nested ones, and much of this paper concentrates on strictly nested contexts.

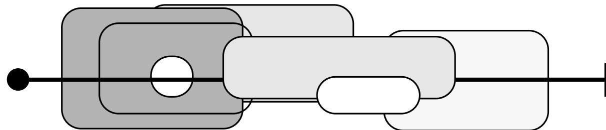


FIGURE 5. Overlapping, non-nested resource contexts

3 Resource-operations within a software design

Resources are used to carry out the operations of the program. Once a resource is obtained, the use made of it will be termed a *resource-operation*, including execution of activities, and requests for additional resources. The interactions with other resources make up the architectural relationships in the software resource architecture.

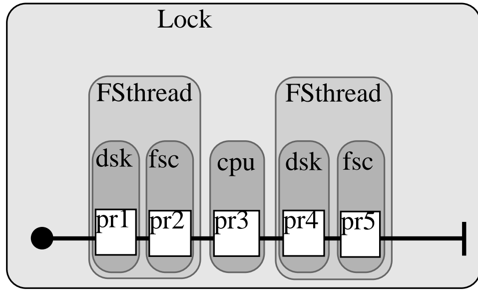
The resource-operation is defined by the scenario fragment that is executed while the program holds the resource. It includes activities and nested resource-operations. For example, the resource-operation of the lock in Figure 4 has the scenario fragment shown in Figure 6(a), including nested operations at the File Server.

The resource-operation definition can be stated in other ways, for example as a UML sequence diagram, as shown in Figure 6(b) (see [2] for a description of the Unified Modeling Language for software). The lock resource context is indicated by requests to a lock manager object, to obtain and release the lock. The part of the sequence diagram showing operations while the lock is held, is shaded lightly in the background. Comparing to part (a), the shaded zone indicates the same lock resource context in both figures. If we extract this part as a separate sequence diagram, it defines the resource operation of the lock, within this scenario. The same information could also be indicated in a UML collaboration diagram.

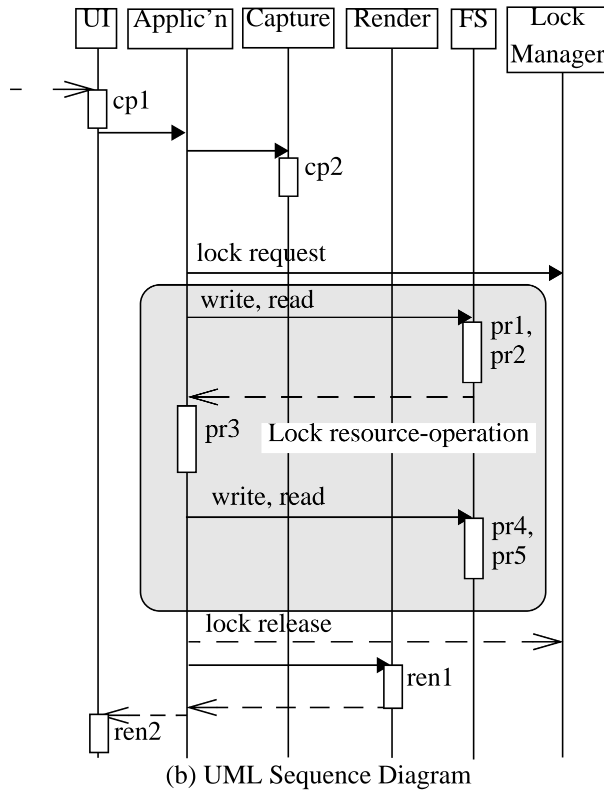
The attributes of a resource-operation that are of interest for resource architecture are:

1. the execution resources of the elementary activities,
2. *strictly nested* resource requests, identifying additional resources that are obtained within this operation, with nested contexts (that is, they are released before the end of this resource-operation). In Figure 4 all the requests are strictly nested.
3. *partly nested* resource requests, identifying additional resources that are obtained during this operation, and not released before the end. In Figure 5 all the requests but on one (indicated by the smallest zone, left blank) are partly nested

During the execution of a program, or of a system of several processes, a given resource is requested many times and different operations are carried out each time. This gives an explosion of resource-operations for the same resource, which conceals the way the system uses the resource, rather than explaining it. To overcome this problem, we can pool these resource-operation instances together.



(a) Lock resource-operation defined by a Use Case Map



(b) UML Sequence Diagram

FIGURE 6. Defining the resource-operation of the Lock in Scenario 2

Pooled resource-operation definitions

There might a large number of instances of use of a single resource like the lock, all with slightly different resource-operations. To represent the general properties of the use of a resource by the system, a pooled resource-operation is derived from all these instances within the collaborations or scenarios of the system. The attributes of the pooled operation are its set of possible requests for other resources. A strictly nested request is represented, for a resource that occurs this way in one or more of the operation instances; a partly nested request is represented for a resource that occurs this way in one or more instances. Thus the same resource may occur in both ways. The processing operations are represented by requests to their execution resources.

Partial pooling: multiple resource-operations for one resource

Instead of pooling all the operation instances into just one resource-operation, they may be partially pooled into groups with significantly different attributes. For example,

- file reads might be identified as one pooled operation, and writes as another.
- one pool could be formed for purely local operations, and another for variations which give rise to remote requests over a network

Partial pooling gives each resource a *set* of pooled resource-operations, each with different interactions. Any request for the resource is made for a particular resource-operation. The decision as to which operations should be pooled together is a matter of judgement, taking into account differences in the patterns of nested requests.

4 Resource architecture

An architectural view of resources, similar to that for software architecture, can be obtained by viewing the resources as components, the requests in the pooled resource-operations as interactions or connectors, and the resource-operations as ports. This gives a diagram such as Figure 7. Interactions are typed according to whether they give strictly nested contexts or not; strictly nested contexts give a call-return or synchronous kind of interaction, indicated by filled arrowheads. Partly nested requests resemble an asynchronous interaction and are indicated by open arrowheads. Since all the requests in Figure 7 are nested, only filled arrowheads appear.

Figure 7 shows the program thread with a single operation to run the program, and the lock with a single operation to run the subscenario discussed above. Each operation has an in-port to connect to requests, and an output where the operation makes requests. The File Server is shown with two resource-operations and a port for each. The box representing the file server is partitioned for readability, so that each operation has a sub-box and its own in-port and out-port.

The device resources (UI, Main-CPU etc.) are also shown, which is optional, but in this case connects to the previous discussion and diagrams. They have execution operations which are not labelled separately.

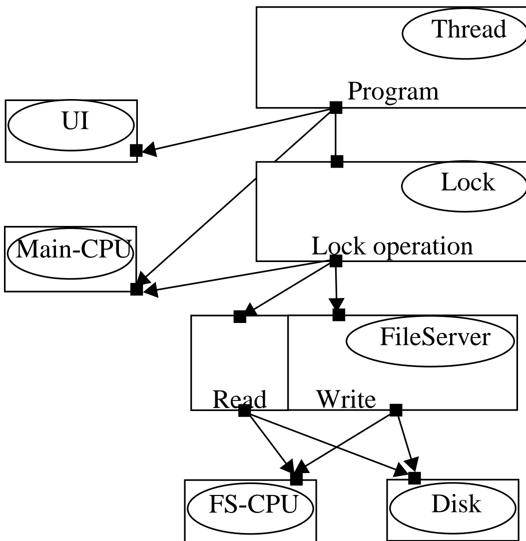


FIGURE 7. Graphical notation for Resource Architectures, related to Scenario 2, showing resource-operations Read and Write for the File Server

The derivation of the architecture diagram may use an intermediate step, which is illustrated in Figure 8 for the same system and scenario. A sub-scenario is shown for each resource-operation, with dashed-outline boxes for the activities. The activities are not part of the architecture notation, but are shown as a stepping-stone to capturing the interactions in the resource-operations. The shaded resources, activities, and request arrows indicate the activities which are active in each operation, and the requests which are at present satisfied, to execute activity *pr1* in Scenario 2. The disk is the active resource and is reading data, preparatory to processing it. The top-level Thread resource is in its *processimage* activity; the Lock resource is in its *read* activity, the file-server resource is in the *read* activity of its Read operation, and the Disk is actually reading the data. The resource context is indicated by the shaded resources, activities and request arrows.

The architectural notation for resources in Figures 7 and 8 conveys more information than the nested contexts shown in Figures 2 to 4. We can see the roles played by a resource across the system, in combination with different other resources; We can see the different contexts within which a given resource may operate; the pooling of resource-operations gives a picture that includes many different nesting patterns. The architectural notation concentrates on the resources, where the nested context notation focuses on the activities.

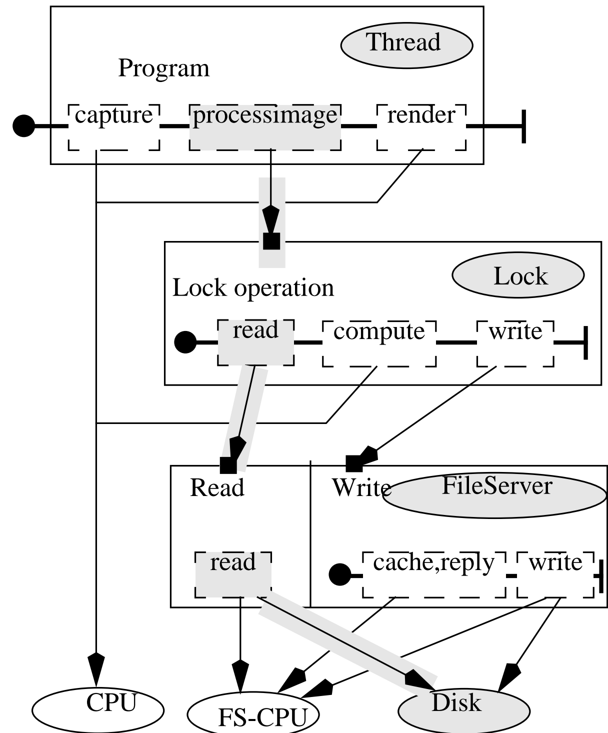


FIGURE 8. Resource architecture with added activity detail in every resource-operation, and with highlighting of the resources, activities and interactions for activity *pr1* of Secnario 2

Additional architectural properties

The basic elements of resource (as a component), request (as a connector or interaction) and resource-operation (as a port, or rather a pair if ports) have been described so far. Additional properties may be included. For instance, a request may simultaneously demand a certain set of resources to be provided as a single atomic permission, either all or none. Centralized lock managers for databases handle requests like this, for a set of locks. The high-level view of a connector abstraction in the ACME architecture description language (ADL) is sufficient to capture such a combined request.

Formal description

A formal description of the resource architecture described here can be recorded using an ADL like ACME [14]. The standard elements of ACME represent the elements of a resource architecture as follows:

- ACME *components* represent resources
- *connectors* represent requests,

- *ports* represent resource-operations, with an in-port and an out-port for each operation,
- *roles* capture the requester and acceptor role for a request

The ACME concept of an *architecture family* could be used to create a sub-language for resource architectures.

Architecture parameters

It may be useful to add parameters to a resource or a resource-operation. Parameters can be used to define the multiplicity of a resource (a multiprocessor, or a multi-threaded task, for instance, or a set of replicas of a server); or to represent the intensity of the dependency of an operation on its requests made to other resources (for instance the average number of times each other resource is requested, or the average execution time of a processing operation).

5 Styles of resource architecture

Characteristic classes of resource architecture are used in certain areas of application, corresponding to styles of software architecture. Four of these, called layered, restart, asynchronous, and pipeline, will be identified here.

- *Layered style:* A common class of systems provides resources and services on demand. A request for a service causes the program to add the appropriate resources to its context, and execute the service, then release the resources.

This is a kind of client-server resource style, with strictly nested resource contexts, as just described. Outer contexts are at a higher layer and inner ones at a lower layer. Lower level resources are held for shorter time spans, nested within the holding times of higher level resources. The “bottom” or innermost resource in any context is normally the device that is actually executing the current operation.

- *Restart style:* In real-time systems in which deadlines must be met every time, deterministic resource behaviour is obtained by restarting the resource acquisition for each task. This style is widely used in automatic control systems and signal processing. The program is divided into distinct “tasks” to execute each of the activities. Deadlines are enforced by the scheduler, backed up with calculations of schedulability, and mechanisms like priority inheritance (see, e.g. [17]). Resource requests are restarted from scratch for each task; the task scheduler allocates the necessary resources to a task before it is launched, and they are released when it ends.

We may call this a “restart” or “operation-centred” resource style; a sequence of separate non-overlapping resource contexts are created. Figure 9 gives an illustration

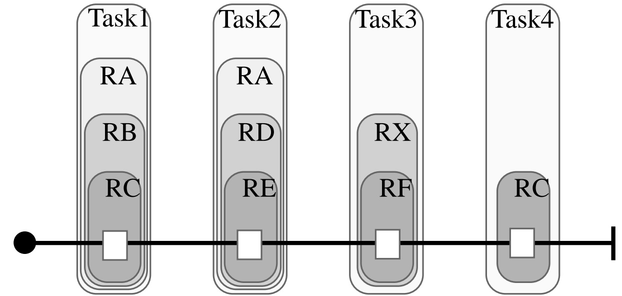


FIGURE 9. Separate resource context for each activity

in which each task has its own operating system thread, and a physical resource (RC, RE, or RF in Figure 9), and each task also has its own collection of logical resources (RA, RB, RD, RX). Even though RA is used again in Task2, it is released after Task1 and then reacquired; this is in contrast with the case in Figure 4.

- *Asynchronous style:* each interacting task has (at least in principle) a self-contained set of resources, so at the task-to-task level the system is idealized as cooperating machines, without shared resources.

This style is used in telecommunications processing, by using asynchronous messages, providing effectively infinite threads for each process, storing data in main memory, and restricting all interactions to be by messages. In this style shared data would be managed by one of the tasks, and locking of shared data is either avoided somehow, or is minimized and treated as an exception to the architecture.

The asynchronous style has something in common with the restart style, in that both attempt to separate the resource contexts of operations. This is due to a shared concern to control response times. However the asynchronous style is adapted to situations in telecommunications which have more variety of sequence and interactions, less predictable behaviour and less strict time constraints.

- *Pipeline style:* Pipelines are widely recognized and used, in hardware and software, and in their classic form they are a “one-at-a-time” resource style. However they may also appear in a generalized form as a sliding series of overlapping contexts, in which resources are acquired in some order and released later in the same order, while operations are carried on. As an example, a file may be opened in one stage and closed in some later stage, with ownership passed along the pipeline.

6 Resource architecture in performance analysis

In the three-view model for software performance engineering described in [33] there were views of Maps (corresponding to software architecture), Paths (corresponding to scenarios), and Resources, which are described by performance models. Entities in the software may appear in one, two or in all three views. The relationships between the views were described in [33]. The resource architecture is a bridge between the software architecture (Map) and the Resource view.

In a different approach, Hofmeister, Nord and Soni [9] describe logical and module views which are types of Map view, and an execution view which takes some account of resources. However the Resource Architecture described here is different, in that it may have a significantly different structure from the software architecture which hosts it.

Resource models may be used to analyse performance. Classical performance models described resources which are used one at a time, and are represented by servers with queues for jobs that are requesting service. These models cannot represent software resources, so extended queueing networks were introduced for this purpose, for example to represent critical sections [16].

A form of extended queueing model for software performance modeling, called layered queueing networks (LQNs), has been developed based on the ideas in Figure 8. The models and tools have been developed by several authors and are described in [31][32][21][6][7][20]. An LQN determines the delay in waiting for all resources, at every level in the layered hierarchy. It accounts for how the holding time of a resource includes waiting and holding for lower level resources. LQNs have acyclic request graphs to avoid resource-based deadlock. That is, because two concurrent programs within the same architecture request their resources in the same order, they cannot deadlock in a situation where each is waiting for a resource already held by the other.

Applications of LQNs include client-server systems, web servers [5], transaction processing [10], distributed databases [26], connection management [11] and "intelligent network" servers in telecommunications [28].

A single-layer LQN, such as would arise from Scenario 1 in Figure 3, is just an ordinary queueing network model. The Program resource in Figure 3 is the customer or customers, and the other resources are the servers.

Our resource architecture model can exploit additional types of interactions defined in LQNs, beyond those described in the previous section. These include:

- non-blocking requests, in which the execution path goes on to a new context when it releases a resource, rather than back to a previous one,
- early release of the requester, while the requested resource continues to be used,
- forwarded requests, which are asynchronous at a low level while holding some set of higher level resources.
- multiple copies of a resource, all identical and managed as a resource pool.

These will now be discussed in greater detail.

Resource pipelines and non-blocking interactions

In a classic pipeline a package of data is passed from one resource to the next, as a buffer or message or file. As it arrives at the next stage it releases the one before. The resource-operations are exactly mapped to the pipeline resources.

Passing data like this will be termed a *non-blocking interaction* between the resource operations. Non-blocking interactions are not limited to pipelines; they also occur when a series of resources is triggered one at a time to work on a job, in any order.

Also, a chain of non-blocking interactions can occur in the midst of a layered system, with some blocked resources also being held throughout the sequence. In this case we may interpret that the request to the first resource in the chain is being forwarded through the sequence, and these are called *forwarding interactions*. They are remarkably common, for example where an input thread dispatches requests to a set of worker threads.

Early release of the requester

We say a requester is *released early* if the resource-operation making a request can resume while the requested resource is still busy. The two resources can then be active in separate concurrent resource contexts, which increases the system concurrency levels.

In one kind of early release interaction, when the requester releases a resource R the execution path splits and an independent resource context is started up, based on R, in parallel with the continuation in the context of the requester. This pattern of resource behaviour compactly captures a fairly common behaviour, for instance:

- a server returns a result to an RPC client, and then does any work which is not in the critical path of the reply, such as buffer clean-up, or logging,
- a pipeline stage accepts its input from a blocked upstream stage (perhaps using a shared buffer), and then releases the upstream stage and continues on by itself,

- a task is handed off to a server to be performed independently.

In layered queueing terminology the part of the resource-operation after the early release is called a “second phase”.

Resources with delayed release

A common pattern of resource holding, which breaks the layered structure, makes an interesting study. Consider a layered system with one exception, a resource which is obtained in a deeply layered context, and retained when other resources “above” it are released. For instance, a system might obtain rights to a buffer on a remote system, through the use of remote resources which are then released, and then later pass the buffer rights to an agent on the remote system, to use. The agent might then use the buffer and then pass it back, release it or pass it on. The resource contexts for such a system are shown in Figure 10, and we can see that they are not nested. This thoroughly breaks the layered resource pattern.

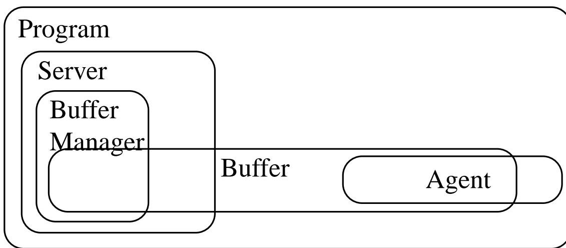


FIGURE 10. Delayed release of a Buffer resource, with non-nested resource contexts.

If resources are layered except for a subset, the subset can be shown as exceptions. For one isolated delayed-release resource, the request and entry into the resource-operation are just as before, but the resource-operation completes without releasing the resource; this must be indicated as a special class of resource. Then the release is a separate interaction, which could be indicated graphically by a special class of arc. Figure 11 shows a Buffer in such a special class of resources, labelled (DR) for delayed release, and the release interaction indicated by an arc labelled Release. There could be multiple alternative release points. A weakness of this notation is that the scope of the resource contexts covered by the Buffer is not clearly identified.

Multiple copies (multiple threads)

A resource may be provided in multiple copies. For instance, the buffer described just above would normally be one of a pool all managed by the Buffer Manager. A server process may be multi-threaded; a processor may be a multiprocessor. Multiplicity should be a parameter of a resource. Sometimes a resource is multiple without any limit, for example a server process which creates a thread per request.

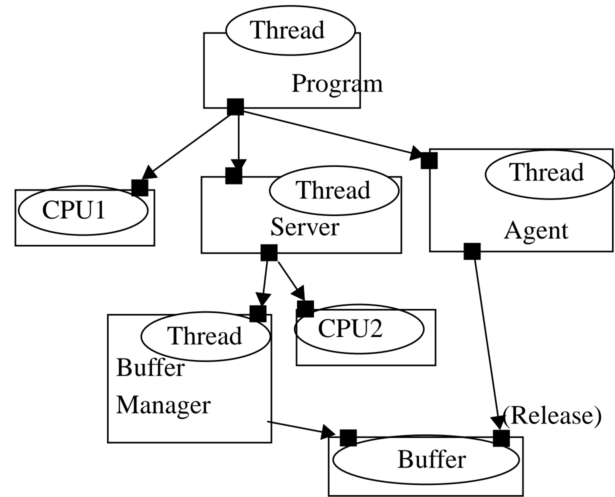


FIGURE 11. Delayed-release resource shown as an exception within a layered architecture

7 Uses of resource architecture

The resource architecture, and the parameters of the resource-operations, governs the performance of the system. Models based on the architecture can be used for evaluation. This has limitations deriving from the degree of abstraction in the architecture; if the architecture is abstract and misses some fine-grained resources, a prediction based on it will miss the effect of those resources. While this could be regarded as a kind of modeling error; it is an inevitable aspect of *abstraction*. The prediction is correct at that level of abstraction

Resource deadlocks

A key architectural property is the existence of cycles in the resource architecture, when it is written as a directed graph with nodes for resources and directed arcs for requests. A cycle indicates the possibility of a resource request deadlock. An acyclic graph with only nested requests can be arranged in layers, with the execution resources in the bottom layer and the application program threads at the top.

Resource overhead

Acquiring and releasing resources incurs overhead which is part of every resource-operation. Fine-grained resource manipulation can cause explosive overhead costs. A nice example is data access under the Simple Network Management Protocol (SNMP), where the protocol states

that every remote value is retrieved separately from a Management Information Base (MIB). This can cause heavy data traffic to be visible in the parameters of the resource architecture [27].

Software bottlenecks

One consequence of a software architecture may be a software resource bottleneck. Layered queuing has been used to investigate this phenomenon, as it relates to process threads [19]. A process thread remains "busy" when it is blocked, waiting for an event or message or reply from some other process or device. A typical source of blocking is waiting for disk I/O to complete. While one thread is blocked, another one could be using the processor, if there is one ready to run. An insufficient thread pool is an example of a software bottleneck, which can relieved by changes (more threads) which are entirely in the software. Another cause of resource starvation is long resource holding times, due to lengthy operations or congested resources at lower levels (e.g. thread starvation because of congestion in the file server). In [19] a measure of "bottleneck strength" was described, to identify where the cause of the bottleneck is located.

Abstract resource architecture patterns

Resource architecture can be the same in systems with very different kinds of resources. Similar software bottlenecks due to logical resources occur at flow control windows (provide a larger window), at a locks (provide finer-grained locking) and even with layered hardware resources, for instance a bus bottleneck when the bus is used to access memory and various interfaces [18]. Patterns in resource relationships that lead to bottlenecks, and strategies for relieving them, have common forms in very different kinds of system. We can identify resources by their roles, and apply similar cures to resource problems in a wide variety of systems.

Complex systems of multiple programs

A resource architecture can also be derived for a *set* of programs which interact at certain resources. This is important for a distributed program which is implemented as a set of collaborating processes, and it can also be used to understand the interactions of separate applications that happen to share software services.

8 Relationship to software architecture

In a given system, how do the resources relate to the software architecture? We can look at this question in three ways: resources as an emergent property, software deliberately designed around its resources, or resources deliberately kept orthogonal to the software structure. There is also the real-time separated-resource-context case, where there is effectively no resource architecture as such.

Emergent resource architecture

The software architecture is determined using a variety of relevant criteria, including performance and resources. A wide-ranging discussion of methods for creating and evaluating architectures is given by Bass, Clements and Kazman [1], and real-time system software architecture evaluation is addressed directly by Kazman et al. in [12]. Some resource issues are dealt with explicitly but other resources for controlling data access, for storing temporary data, for concurrent threads, and so forth may just accumulate from the totality of considerations. In this way a resource architecture emerges, and it may or may not have a recognizable structure. How can it be recognized and extracted?

Kazman and Carriere have considered a similar question for software architecture, and found the relationships from static analysis [13]. For resources, it is clear that scenarios or traces must be analyzed to identify resource demands and holding times of higher level resources. This is the basis of classic software performance engineering recommendations by Smith [29], for instance. However Smith concentrated on demands from hardware devices, and gave only limited assistance for dealing with logical resources and concurrent execution. In [30] Smith and Williams considered software architecture and performance, but the architecture was at a very fine-grained level (data objects) and mainly affected the hardware demands. Hrischuk et al. [10] have described an approach for finding resource contexts and layered performance models from traces, even when objects have been dynamically created and linked.

The resource issues that emerge during analysis may also be due to hardware resources. In [12], performance issues which arose in evaluating a real-time embedded software architecture centred around a hardware limitation (a channel bandwidth).

Emergent resource architecture partly follows software structure. Concurrent processes involve resources, for instances. Semaphores can be identified, but the requests may be buried in low level modules even though they govern higher-level operations by other modules. Thus the scope of a resource-operation may be difficult to determine. Buffer resources pose similar questions.

"Resources first" development

Systems with critical performance requirements, that are not amenable to separate-context design may be designed around the critical resources. Internet routers come to mind as an example, with the routing table as one critical resource.

In designing reactive software systems (systems where the function is mainly to respond in a timely way, and in the correct order, to external events) Selic et al. have insisted that architecture should not only identify components and interactions, but must also describe behaviour [23]. This is certainly also true where resources are considered first. It is implied by the central position of resource-operations and their interactions, in the architecture. Scratchley and Woodside have considered concurrency-related architecture decisions within an integrated scenario specification in [22], and have evaluated substantial alternatives in software architecture for a group communications system. Some ideas for generating architectural alternatives around performance concerns were described in [33].

A general approach is to develop the resource architecture first, from an analysis of scenarios, then to estimate budgets for operation times, validate the performance measures on the basis of the budgets, and finally go on to develop all the other aspects of the software within the budgets. This process mimics the way projects are managed to fit within financial budgets, and allows for iterations and adjustments as problems are revealed. This is a subject of current research.

Resources orthogonal to software

This name is applied here to systems in which the software design avoids resource commitments, so it can be deployed in many different situations, with different resources. This approach appears to be implicit in many theoretical ideas of distributed systems. Ideally, resources can be completely ignored in the software, and included in a configuration step which specializes the system to a particular deployment option.

In practice, there are still *resource roles* in the design, or implied by it. They may be employed only in some versions of the system. For instance semaphores to protect data shared by multiple threads are not needed in a small-scale single-threaded deployment. The resource architecture is a property of the deployment, and is effectively isolated from the software architecture. However the semaphore programming has to be provided in the system, even if its use is made optional.

In this situation there may be many resource architectures that could be used with a single software architecture, and they may be quite different. For instance a series of operations in a subsystem might be configured as a resource pipeline in one deployment and a hierarchical master-slave style in another. The possibility exists of optimizing the deployments within a general plan, and some of these issues were explored in a recent study of scalability of software architectures [11].

The programming to support multiple resource configurations is likely to be complex, which is a negative aspect of this concept.

Self-contained tasks

Concurrent software tasks may be designed to use a self-contained set of resources, at least while it is active. Even the processor is viewed as not being shared while the task is active, by requiring that the task runs to completion. This is a very “resource-aware” form of software design and can be used for hard-real-time systems or wherever resources are critical. However if a task must run to completion it limits the use of modern schedulability-based methods for achieving deadlines, since these are based on task pre-emption. If run-to-completion can be relaxed, pre-emption may be permitted by a “compatible” task that uses a different set of resources.

Separated contexts and central resource management

Deadline driven systems with the “restart” style of resource architecture tend to have a flattened style of software architecture, as well as self-contained resources. Shared server tasks with blocking requests, for example, are absent or discouraged. Central control by the system scheduler establishes control over timing of execution of operations, but limits the system’s applications and makes it sensitive to changes. As examples, the design may depend critically on a clock rate, or on how many functions have to be executed in an execution cycle. Once any aspect of such a tightly controlled solution breaks down the system must be reconsidered from the beginning. The composition of systems into larger systems has to be done with great care and may not be possible. As a result, this approach does not appeal to designers of systems in which it is not essential, for instance in telecommunications.

9 Conclusions

A concept of resource architecture has been described, which applies to software resources as well as to hardware. There are resource-operations attached to each resource, and resource interactions between the operations. A resource architecture can be determined for any software system, and often has a layered structure. The layered form is backed up by a performance modeling methodology called layered queueing. The software resource architecture can also be used to analyze for resource deadlock possibilities, for insight into the range of resource contexts that the system will use, and for insight into the interactions between systems that are designed separately but which share some software resources.

Resource architecture may be imposed on a system from the beginning by its designers, or may be identified

as an emergent property of a mature system. Identifiable styles of resource architecture include pipelines, hierarchies and layers, and operation-centred styles. A chaotic architecture, with resource relationships that have no particular structure, may be subject to inefficient resource use and to resource deadlock.

The resource architecture is in general not simply a reflection of the software architecture, but has separate entities and relationships. However, in two demanding classes of application, in control and in telecommunications, there is a strong correspondence between the two. The restart and asynchronous architecture styles described above force them to line up, more or less, and also constrain the software style. Deeper study is needed into the general question of alignment of the two aspects of architecture, and the need in some systems to give primacy to an effective resource architecture.

For understanding performance issues, and adapting designs to solve performance problems, software resources will usually be considered together with hardware resources, in a *system resource architecture*, which can also be described by the approach of this paper. In fact the examples given here include some hardware resources, and hardware resources can also be layered. The integrated study of software and hardware resources is essential for understanding the operation of the system. However a given software system is likely to be deployed in many different environments, so its software resource architecture should be understood first and then considered as a component in many system architectures.

Acknowledgements

This research was supported by the Natural Sciences and Engineering Research Council of Canada through its program of Research Grants.

References

[1] L. Bass, P. Clements, R. Kazman, *Software architecture in practice,* Addison Wesley, 1998

[2] G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesely, 1999.

[3] R.J.A. Buhr, "Use Case Maps in Scenario-Based Design", IEEE Trans. on Software Engineering, vol. 24, no 12, Dec. 1998, pp 1131-1155.

[4] L. Cheung, B.A. Nixon, E. Yu, "Using non-functional requirements to systematically select among alternatives in architectural design", *Proc. of First Int. Workshop on Architectures for Software Systems*, April, 1995, pp 31-43.

[5] J. Dilley, R. Friedrich, T. Jin, J.A. Rolia, "Measurement Tools and Modeling Techniques for Evaluating Web Server Performance", *Proc. 9th Int. Conf. on Modelling Techniques and Tools*, St. Malo, France, June 1997

[6] R.G. Franks, S. Majumdar, J.E. Neilson, D.C. Petriu, J.A. Rolia and C.M. Woodside, "Performance Analysis of Distributed Server Systems", Proc. Sixth International Conference on Software Quality, Ottawa, Canada, October 28-30, 1996, pp. 15-26.

[7] Greg Franks, Murray Woodside, "Performance of Multi-level Client-Server Systems with Parallel Service Operations", *Proc. First Int. Workshop on Software and Performance (WOSP98)*, pp. 120-130, Santa Fe, October 1998.

[8] D. Garlan, R.T. Monroe, D. Wile, "ACME: An Architecture Description Interchange Language", Proc. CASCON 97, Toronto, pp 169-183.

[9] C. Hofmeister, R. Nord, D. Soni, *Applied Software Architecture*, Addison-Wesley, 1999.

[10] C.E. Hrischuk, C.M. Woodside, J.A. Rolia, "Trace-Based Load Characterization for Generating Software Performance Models", IEEE Trans. on Software Engineering, v 25, n 1, pp 122-135, Jan. 1999.

[11] Prasad Jogalekar, Murray Woodside, "Evaluating the Scalability of Distributed Systems", IEEE Trans. on Parallel and Distributed Systems, to appear in 2000.

[12] R. Kazman, M. Klein, P. Clements, "Evaluating Software Architectures for Real-Time Systems", Annals of Software Engineering, Vol. 7, 1999, 71-93.

[13] R. Kazman, S. J. Carriere, "Playing Detective: Reconstructing Software Architecture from Available Evidence", Journal of Automated Software Engineering, 6:2, April, 1999, 107-138.

[14] A. Kompanek, "Modeling a System with ACME", ABLE Group report, School of Coimputer Science, Carnegie-Mellon University, 1998. Available at http://www.cs.cmu.edu/able

[15] O. Lassila, R. R, Swick (eds), Resource Description Framework (RDF) Model and Syntax Specification, W3C Recommendation 22, World Wide Web Consortium, Feb. 1999.

[16] E. Lazowska, J. Zahorjan, S. Graham, K. Sevcik, *Quantitative System Performance*, Printice Hall, 1984.

[17] S. Levi, A. K. Agarwala, *Real Time System Design,* McGraw-Hill, 1990.

[18] P. Maly, C.M. Woodside, "Layered Modeling of Hardware and Software, with Application to a LAN Extension Router", Proc. Performance Tools 2000, Chicago, March 2000.

[19] J.E. Neilson, C.M. Woodside, D.C. Petriu and S. Majumdar, “Software Bottlenecking in Client-Server Systems and Rendez-vous Networks”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 776-782, September 1995.

[20] S. Ramesh, H.G. Perros, “A Multi-Layer Client-Server Queueing Network Model with Synchronous and Asynchronous Messages”, *Proc. of First Int. Workshop on Software and Performance (WOSP98)*, pp. 107-119, Oct. 1998

[21] J.A. Rolia, K.C. Sevcik, “The Method of Layers”, IEEE Trans. on Software Engineering, v 21, no. 8, pp 689-700, August 1995.

[22] C. Scratchley, C. M. Woodside, “Evaluating Concurrency Options in Software Specifications”, Proc 7th Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecomm Systems (MASCOTS99), College Park, Md., pp 330 - 338, October 1999.

[23] B. Selic, G. Gulleckson, P. T. Ward, Real Time Object Oriented Modeling, publisher, 1994.

[24] M. Shaw and D. Garlan, *Software Architecture*, Prentice-Hall, 1996

[25] M. Shaw, Sufficient Correctness and Homeostasis in Open Resource Coalitions”, WORKSHOP at Nth Int Conf on Software Engineering (ICSE2000), Limerick, June 2000.

[26] F. Sheikh and C.M. Woodside, “Layered Analytic Performance Modelling of a Distributed Database System”, *Proc. 1997 International Conf. on Distributed Computing Systems*, May 1997, pp. 482-490.

[27] F. Sheikh, J.A. Rolia, P. Garg, S. Frolund, A. Shepherd, “Performance Evaluation of a Large Scale Distributed Application Design”, *Proc. of World Congress on System Simulation*, Singapore, September, 1997 .

[28] C. Shousha, D.C. Petriu, A. Jalnapurkar, K. Ngo, “Applying Performance Modelling to a Telecommunication System”, Proc. of First International Workshop on Software and Performance (WOSP98), pp. 1-6, October 1998.

[29] C.U. Smith, Performance Engineering of Software Systems, Addison-Wesley, 1990.

[30] C.U. Smith, L.G. Williams, “Performance Evaluation of Software Architectures”, *Proc First Int. Workshop on Software and Performance (WOSP98)*, pp. 164-177, October 1998

[31] C.M. Woodside, “Throughput Calculations for Basic Stochastic Rendezvous Networks”, Performance Evaluation, v 9, no 2, April 1989.

[32] C.M. Woodside, J.E. Neilson, D.C. Petriu and S. Majumdar, “The Stochastic Rendezvous Network Model for Performance of Synchronous Client-Server-Like Distributed Software”, IEEE Transactions on Computers, Vol. 44, No. 1, January 1995, pp. 20-34.

[33] C.M. Woodside, “A Three-View Model for Performance Engineering of Concurrent Software”, IEEE Trans. On Software Engineering, Vol. 21, No. 9, pp. 754-767, Sept. 1995.

[34] C.M. Woodside, “Software Resource Architecture and Performance Evaluation of Software Architectures”, Proc. 34th Hawaii Int. Conf on Systems Sciences, Jan. 2001.