A Threaded/Flow Approach to Reconfigurable Distributed Systems and Service Primitives Architectures

Lester F. Ludwig

Bell Communications Research
Red Bank, New Jersey

ABSTRACT

This paper discusses a methodology for managing the assembly, control, and disassembly of large numbers of independent small-scale configurations within large-scale reconfigurable distributed systems. The approach is targeted at service primitives architectures for enhanced telecommunications networks, but can apply to more general settings such as multi-tasking supercomputers and network operations systems.* Study of the methods presented here was a key motivation in founding the Bell Communications Research *Integrated Media Architecture Laboratory (IMAL) [1].*

The Threaded/Flow approach uses data-flow constructs to assemble higher level functions from other distributed functions and resources with arbitrary degrees of decentralization. Equivalence between *algorithms* and *hard and virtual resources* is accomplished via threaded-interpretive constructs. Function autonomy, concurrency, conditional branching, pipelining, and setup/execution interaction are implicitly supported. Some elementary performance comparisons are argued.

This work is motivated by telecommunications applications involving coordinated multiple-media in open architectures supporting large numbers of users and outside service vendors. In such networks it is desired that services may be flexibly constructed by the network, service vendors, or by users themselves from any meaningful combination of elementary primitives and previously defined services. Reliability, billing, call progress, real-time user control, and network management functions must be explicitly supported. These needs are handled with apparent high performance by the approach.

1. INTRODUCTION

This paper discusses a methodology for managing the assembly, control, and disassembly of large numbers of independent small-scale configurations within large-scale reconfigurable distributed systems. The methods are targeted at service primitives architectures for enhanced telecommunications networks discussed at the end of the paper. However, the approach also applies to more general settings such as multi-tasking supercomputers and network operations systems as well as very general settings involving a functionally-distributed system which is reconfigurable.

The approach employs a functional substrate which equates hard resources, virtual resources, and algorithms. This permits an existing service (typically an algorithm) to be freely combined with resources of different types in a unified way; in fact, services and resources are managed and allocated in the same basic manner. A system for fetching service descriptions from databases and constructing data-flows between named entities provides the basic foundation for the method. A named entity is either a resource (controlled by a resource manager) or another algorithm which may itself control other algorithms and resources. A given entity may exist and execute its tasks anywhere in the network and can itself be of a distributed nature. Branching, concurrency, pipelining, and arbitrary degrees of decentralization are naturally supported. It is straightforward to pipeline the setup and execution phases of the data-flow so that parts of the data-flow which are already set up may execute as remaining parts are themselves being set up; this permits conditional setup of a data-flow as a function of events encountered in earlier parts of its execution. Since an existing stand-alone service can be linked to (i.e., peer connections) or included in (i.e., hierarchical connections) another service, the system acts very much like a threaded interpretive language. It is also straightforward to replace service description fetches from databases with user downloads, permitting user definition of services from service primitives. Filters to preserve integrity of the network will require straightforward design but are not explicitly considered here to limit the discussion.

The approach presented here was developed for use in ISDN prior to the CCITT standards that established ISDN as it is recognized today. It differs somewhat in style from conventional trends in distributed processing since the key application concerns control in the context of extremely large-scale telecommunications (campus-wide, city-wide, or nation-wide) networks. The emphasis is on executions of well-defined control procedures rather than execution of arbitrary user programs.

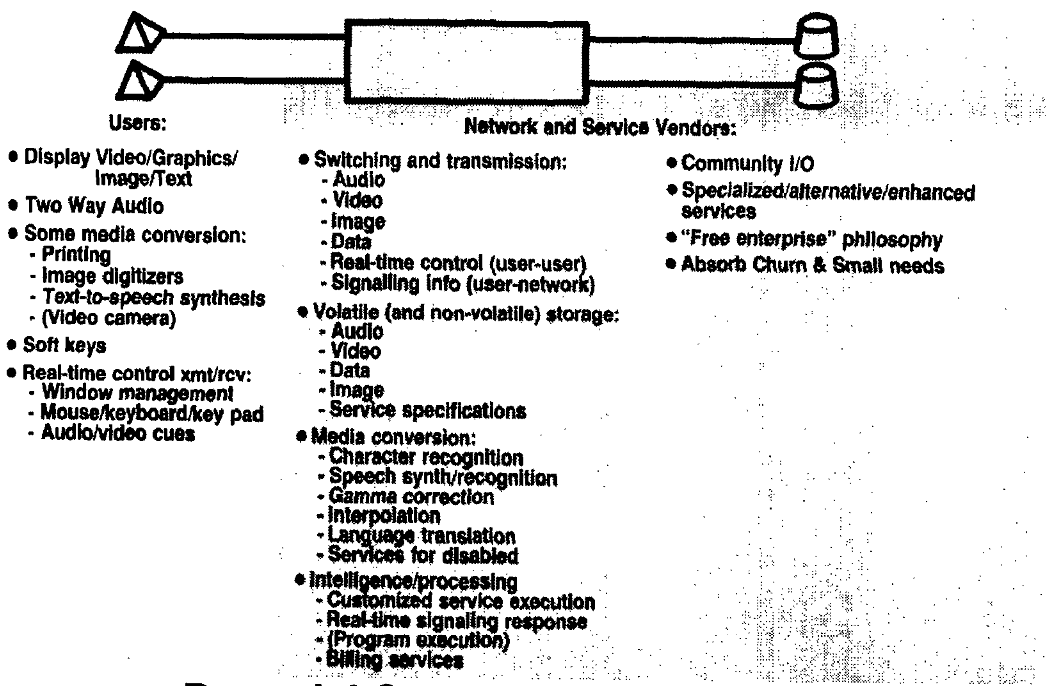
1.1 Need For Service Primitives In Advanced Telecommunications Networks

The main reasons for resorting to a service primitives approach in a telecommunications network are the potential combinatoric complexity of service details, evolution of services in un foreseen directions, and a need for simple ways of linking existing services and and other functional elements to create new expanded services.

* *Most of this work was performed in 1982 and refined from 1983 - 1986 while the author was at U. C. Berkeley.*

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1988 ACM 0-89791-245-4/88/0001/0306 $1.50


Users:
• Display Video/Graphics/ Image/Text
• Two Way Audio
• Some media conversion:
- Printing
- Image digitizers
- Text-to-speech synthesis
- (Video camera)
• Soft keys
• Real-time control xmt/rcv:
- Window management
- Mouse/keyboard/key pad
- Audio/video cues
Network and Service Vendors:
• Switching and transmission:
- Audio
- Video
- Image
- Data
- Real-time control (user-user)
- Signalling info (user-network)
• Volatile (and non-volatile) storage:
- Audio
- Video
- Data
- Image
- Service specifications
• Media conversion:
- Character recognition
- Speech synth/recognition
- Gamma correction
- Interpolation
- Language translation
- Services for disabled
• Intelligence/processing
- Customized service execution
- Real-time signaling response
- (Program execution)
- Billing services
• Community I/O
• Specialized/alternative/enhanced services
• "Free enterprise" philosophy
• Absorb Churn & Small needs

Potential Service Complexities

Figure 1

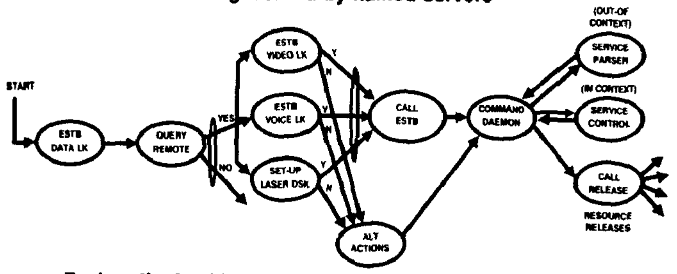
Figure 1 illustrates the potential service combinatorics. Each of the user, network, and service vendor domains is predicting and working toward support of a number of telecommunications features as shown. It is startling to notice how astronomical the potential combinations of "reasonable" user expectations for telecommunications might be. It is impossible to address these sophistications and combinations using the existing methodology of monolithic software generics and hardware upgrades oriented around one additional single-feature service at a time. In addition, it is natural to expect services to become linked or incorporated into others. For example, users may subscribe to automated calendar management services or have one in their own network-accessible systems. A service vendor could link these with a network-provided call forwarding service to form a meeting scheduling service; subscribers could name meeting participants and constraints and the automated service would forward potential dates and places to participants, optimize, and finally distribute word of the meeting time and place as well as supporting documentation using call and mail utilities. Later, another service vendor may link this meeting scheduling service with a travel booking service to automatically arrange transportation. From the user viewpoint, it is natural to expect that such capabilities should be made available from an advanced communications and applications environment.

The highly succinct explanations above illustrate that the key to managing the immense possibilities within future telecommunications networks is a flexible, powerful, and simple-to-use service primitives system. A reasonable set of goals for such a service primitive system include:

1. Span the possible service combinatorics made possible by technology, including the management and linking of simultaneous calls;
2. Support simple ways of linking existing services and other functional elements;
3. Provide monitoring, billing, and reliable failure recovery functions;
4. Provide ways for the network, its users, or outside service vendors to create their own services in an open architecture.

The service primitives solution suggested in this paper uses an architecture for the direct implementation of data-flow descriptions of services. An example of such a description is shown in Figure 2. Each named process can be an elementary *resource function* a *service element* (i.e., raw resource functions with controlling programs), or a full stand-alone service. Because of this, what is viewed as a fundamental "service primitive" can be quite arbitrary: they could be any subset of the possible collections of raw resource functions, explicitly constructed primitive service elements, or existing stand-alone services. This allows a network or :ervice architect to

• A data flow description of transactions and interconnections between entities governed by named servers



• Each entity is either a "process" or a "resource"

• Each process may have subprocess (specified at lower levels)

Service Descriptions

Figure 2

experiment with custom designed collections of service primitives which may be freely modified. Because of this, the suggested approach is actually a generalized service primitives architecture; in fact, it is a *programmable services primitives engine* on a network scale with explicit attention given to architectural, operations, maintenance, administrative, fault-tolerance, failure recovery, billing, and monitoring design issues that are relevant to large-scale network implementations.

Rather than naming specific primitives explicitly as has been suggested in other service primitives methods, this approach only needs to explicitly define and implement the following elements:

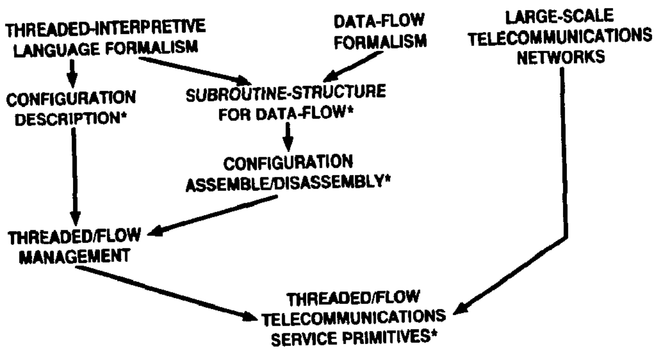
1. Service description database system;
2. Service Execution Coordinators (SECs) and their servers;
3. Resource managers;
4. Control messages and their precise formats;
5. Overall systems management utilities;
6. Monitoring and reporting utilities.

With this substrate, creation of explicit service primitives and services themselves can be done with very short "program-like" files which are easy to create and change. This approach naturally permits easy introduction and evolution of service primitives. It also lends itself well to laboratory experimentation.

1.2 Strategy of the Paper

Figure 3 illustrates the evolution of the ideas as developed in this paper. The development is structured as follows:

1. Drawing from threaded-interpretive language ideas [2], a generic method for constructing functions from resources and previously created functions is developed. The method employs two parts: (1) descriptions of particular configurations within the reconfigurable system and (2) the passing of parameters, files, or software to specific functional elements in the configuration. *This method permits functions to be associated in a well-defined manner with linguistic descriptions, but does not provide a means of handling the actual assembly or disassembly of the functions it concerns.*
2. Threaded-interpretive language ideas are then used a second time; here they are coupled with data-flow concepts [3,4,5,6] to create a specific means of handling data-flow type subroutines. The result is used, subject to the structure of configuration description method just discussed, to create a means for the assembly and disassembly of the configurations.


THREADED-INTERPRETIVE LANGUAGE FORMALISM
DATA-FLOW FORMALISM
LARGE-SCALE TELECOMMUNICATIONS NETWORKS
CONFIGURATION DESCRIPTION*
SUBROUTINE-STRUCTURE FOR DATA-FLOW*
CONFIGURATION ASSEMBLE/DISASSEMBLY*
THREADED/FLOW MANAGEMENT
THREADED/FLOW TELECOMMUNICATIONS SERVICE PRIMITIVES*

Development Of The Control Synthesis. The Author Views Items Denoted (*) As New Work In The Area
Figure 3

3. The two items listed above are united to create an overall method for managing the assembly, control, and disassembly of large numbers of independent small-scale configurations within large-scale reconfigurable distributed systems. The approach permits arbitrary degrees of decentralization. It also permits configurations to be partially assembled, used as such, and then continue assembly/disassembly steps as a function of the outcomes of this usage (i.e., assembly, usage, and disassembly may be freely pipelined). *The development to this point is general, applying to settings such as multi-tasking supercomputers, enhanced telecommunications networks, and network operations systems.* The approach to this point is entitled a "Threaded/Flow" approach for lack of more creative thoughts.
4. The abstract Threaded flow approach is combined with perspectives concerning large-scale telecommunications networks to form a service primitives method for enhanced telecommunications networks. Discussion of full detail is somewhat suppressed here in view of the audience and potential legal and proprietary issues currently under consideration.

Most of the paper focuses on the abstract Threaded/Flow approach. The items listed above are supplemented with remarks on performance, server design, and analytical study. With the exception of the analytically-oriented Sections 6 and 8, the level of this paper is comparable to that of technical introductions to the OSI reference model.

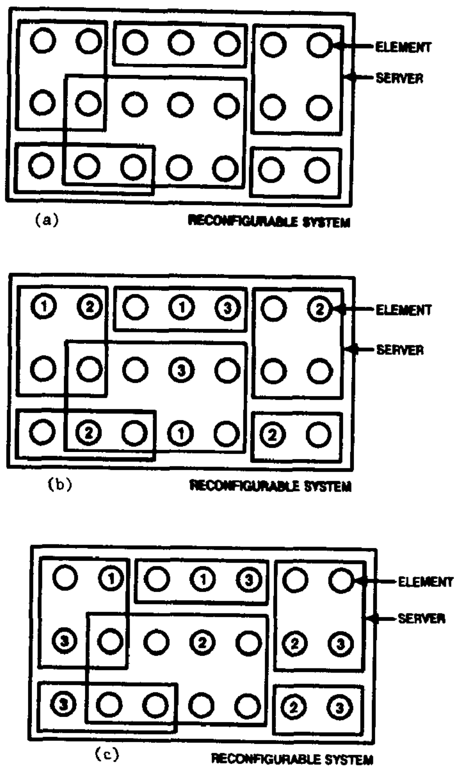
In this paper a number of special terms are defined and used in subsequent discussion. These terms are printed in **bold type** where they are first mentioned and defined.

2. CONFIGURATIONS WITHIN A LARGE-SCALE DISTRIBUTED SYSTEM

Figure 4 illustrates the basic setting, i.e., a large-scale reconfigurable system with distributed **functional elements.** The functional elements are treated as shared resources which are allocated by one or more **servers.** Each server may administer multiple functional elements. The functional elements may be either hard or virtual resources. Servers allocate these resources, in accordance with currently active network management policies, to a specific task on the basis of **resource requests.** In particular, each specific task will require one or more resources which must be arranged and interconnected with information flows. A specific arrangement and interconnection is termed a **configuration.** The large-scale reconfigurable system exists so that configurations can be assembled on request from the pool of available functional elements, applied to serve a specific task, and then dismantled to free resources for subsequent tasks. In particular it is assumed the large scale system can support a large number of such configurations simultaneously and that most of these configurations will be created, operated, and dismantled independently from one another. Figure 4a shows the overall collection of resources and servers, while Figures 4b and 4c illustrate two possible sets of active configurations. The collection of active configurations present at a given instant may be viewed as the **configuration state** of the large-scale reconfigurable system. Hence, Figures 4b and 4c illustrate two possible configuration-states.

2.1 Uniqueness & Operations On Configuration-States

To clarify terminology, it is required that a given resource may only be employed in at most one configurations. Two or more configurations can be **merged** to form a new configuration at which time the original configurations involved are no longer separately recognized as a feature of the configuration-state; i.e., a merger results in a change of configuration-state. Two or



Independent Configurations Within A Reconfigurable System: (a) Setting, (b) A Configuration State, (c) Another Configuration State
Figure 4

more configurations may also be linked together at higher levels but such an arrangement will not be viewed as affecting the configuration-state.

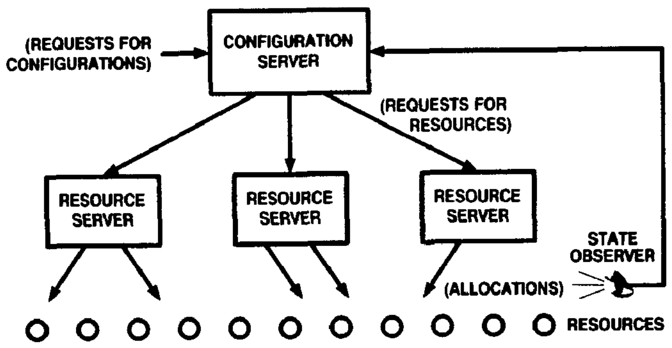
2.2 Configuration Servers

In general, the large-scale reconfigurable system will be able to realize a large number of possible configurations. Also in general, and in most cases by design, the large scale system will be able to support multiple independent copies of the same configuration. Thus, if the admissible configurations are C_1, C_2, ... C_K, there can be n_1 copies of configuration C_1, n_2 copies of C_2, ..., and n_k copies of C_k all active at a given instant, subject to resource limitations:

$$(n_1, n_2, \cdots, n_k) \in \mathbf{L}$$

where $\mathbf{L}$ is some set of admissible configuration states (mathematically, $\mathbf{L}$ is a subset of the collection of non-negative integer-valued k-tuples $\mathbf{Z}_+^k$).

The outside world presents the large-scale reconfigurable system with requests for configurations. Assumed here is that these requests are largely independent of one another and that each configuration is limited to employing only a small fraction of the resources available in the system. These assumptions validate the need for **configuration servers**, i.e., entities which accept or reject configuration requests based on the current configuration state. An accepted request is passed to resource servers in the form of resource requests (which are translated into actual resource allocations by the resource servers). This is illustrated in Figure 5. It is noted that the configuration server may not have full information of the configuration state; more than likely it will work from simplified information provided by the resource servers.



Configuration Server Setting
Figure 5

2.3 Hierarchical Features

The purpose of configurations is to implement higher level functions. That is, each configuration performs a function itself, comparable to the functions provided by the functional elements themselves, but with more sophistication. For convenience call the functions provided by configurations **configuration functions**. At this point two key observations are relevant:

1. Since both configurations and functional elements implement functions, configuration functions and functional elements may be viewed as being equivalent except for the level of sophistication;

2. Just as functional elements may be brought together to form configurations, it is also possible to construct configurations out of configurations themselves.

   These two observations may be combined to yield third and fourth observations:

3. The collection of configuration functions together with functional elements can be used as an extended collection of elements, from which higher level configurations may be created.

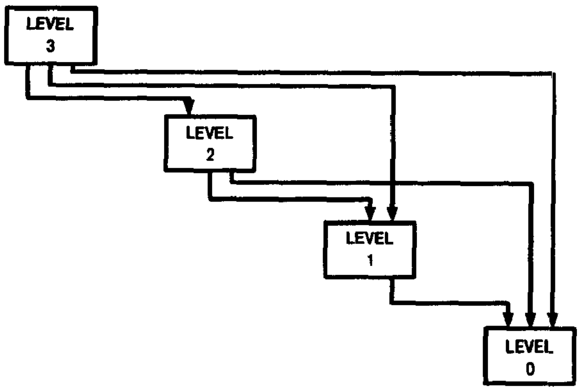
4. This process can be iterated as follows:
   - Define functional elements as "level.0" configuration functions
   - For K = 1, 2, 3.. - define **"level.K" configurations** as configurations created from the pool of resources consisting of "level.K-1", "level.K-2", ... "level.0" configuration functions which include at least one "level.K-1" configuration function.

In this way, just as in structured programming, Lego-blocks,* or an Erector-Set,* raw functions at any level of sophistication can be drawn upon in a unified way to construct more sophisticated configurations.

The nature of the fourth observation is not unlike that of a threaded interpretive language. The resulting possible constructions are illustrated in Figure 6. For the moment a "level.K" configuration can be thought of as being administered by **"level.K" configuration servers**, an artificial concept to be used temporarily.

It is explicitly noted that the layering discussed here results from a general hierarchical construction *and has nothing to do with the seven-layer OSI reference model.*

* American children's toys featuring elements which can be assembled in easily nested modules.



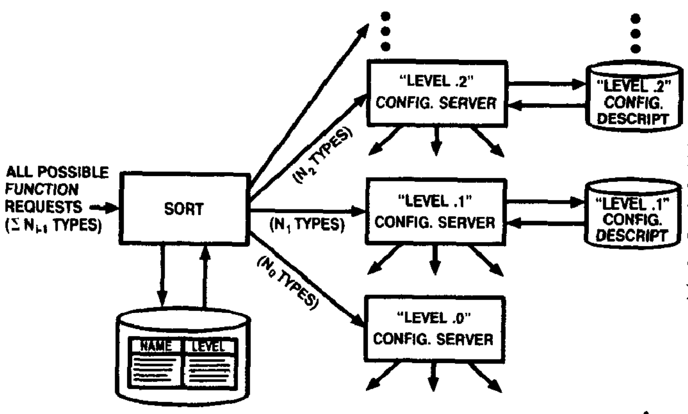
Hierarchy With Triangular Connectivity
Figure 6

2.4 Naming Complexities

For each value of K=0,1,2, · · · let there be N_K different "level.K" configuration functions. Then a fully capable "level.K" configuration server must be internally aware of each of the

$$\sum_{i=0}^{K-1} N_i$$

different configuration functions available to it, yet deal with requests for only N_K types of configuration functions. This suggests an implementation strategy for controlling complexity as illustrated in Figure 7. A simple database recognizes function names and routes requests to configuration servers of an appropriate level. These configuration servers need only worry about a few (i.e., N_K at level K) configuration descriptions. This conceptual structure may be used either directly or indirectly in implementing user interfaces and/or look-ahead pipelining during configuration set-up (to be discussed later).


ALL POSSIBLE FUNCTION REQUESTS ($\Sigma N_{i,i}$ TYPES)
SORT
NAME
LEVEL
(N_2 TYPES)
(N_1 TYPES)
(N_0 TYPES)
"LEVEL .2" CONFIG. SERVER
"LEVEL .1" CONFIG. SERVER
"LEVEL .0" CONFIG. SERVER
"LEVEL .2" CONFIG. DESCRIPT
"LEVEL .1" CONFIG. DESCRIPT

Definition Management
Figure 7

3. CONFIGURATION ASSEMBLY

Configuration functions are an important advantage offered by a reconfigurable system. However, these functions are only available after a configuration has been assembled. Further, unless configurations are disassembled after their use by the requester, resources will be wasted and assembly of configurations will soon be impossible. Thus, the control required for the assembly and disassembly of configurations is of key importance for realization of this concept. In addition, it is naive to believe that large-scale reconfigurable systems could be well centralized, especially if subject to any reasonable amount of evolution or change. Although this paper targets large-scale reconfigurable systems that are decentralized, it is natural to suspect that almost all practical large-scale reconfigurable systems will have to be at least somewhat decentralized. It is here that data-flow like constructs are very useful. As it turns out, the resulting features made possible by decentralized assembly control have some interesting potential performance advantages in many situations which can be well characterized. As a result, potential exists for well-defined design principles concerning degrees of decentralization appropriate for a given situation.

Consider a request for a "level.K" configuration function by a user of the large-scale reconfigurable system. Depending on implementation, the user may or may not specify the precise level of the function. Further, the level may or may not be useful in administering assembly of the requested configuration.

It is assumed that full information describing how each configuration is to be assembled is represented as software files to permit easy evolution and change. The full information describing how each configuration is to be assembled is called a **configuration assembly description** and is assumed to be stored in databases that are made available to configuration servers. Other arrangements are possible, including one where users are free to provide this full description themselves.

The goal is to create a mechanism which simplifies configuration description software as much as possible yet operates with high performance in an environment where configuration functions (at least a "level.0") are distributed. The suggested approach is to use a data-flow like substrate between configuration servers at all levels. This data-flow substrate provides a message-passing system that is:

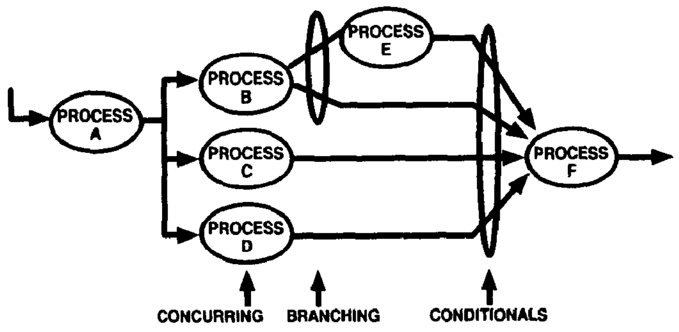
1. Oriented towards the execution of tasks and processes;
2. Supports branching, conditionals, and concurrency;
3. Operates transparently and naturally in either centralized or decentralized environments.

In particular, the degree of decentralization is determined precisely by the configuration assembly description.

3.1 The Data-flow Formalism

The mechanism used for configuration assembly is more precisely a type of "control flow" (see, for example, [3]) since it is intended for control rather than computation. The structure of the configuration assembly mechanism is almost identical to the traditional data-flow formalism which is well documented (see, for example [3,4,5,6]).

The basic features of the data-flow formalism are illustrated in Figure 8. In the Figure, processes are represented by labeled

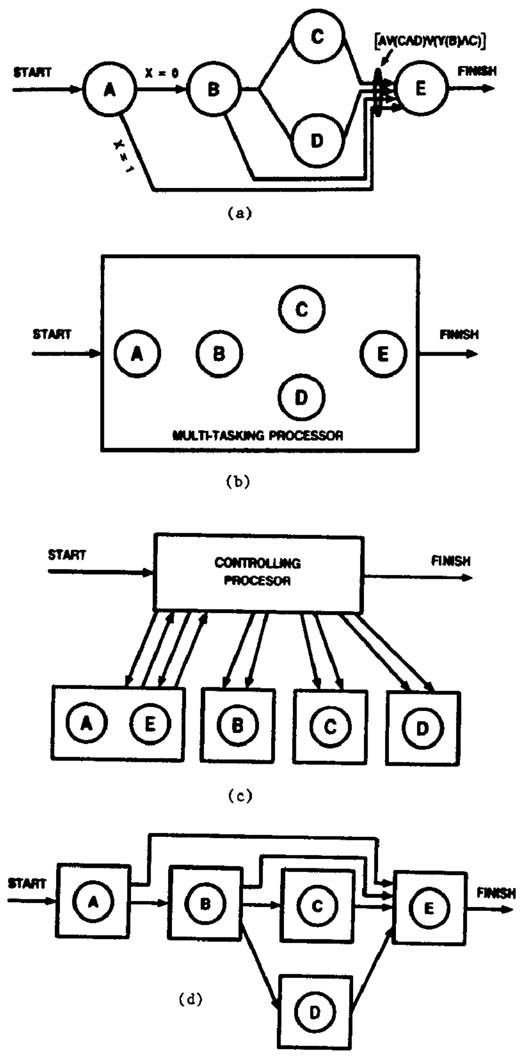


The Data-Flow Formalism
Figure 8

ovals. **These ovals are interconnected** by message-passing paths which are represented by arrows. These paths carry **completion tokens** which are used to signify that the previous process has been completed. These tokens also may carry information created by the previous processes if appropriate. A given process collects these tokens until a logical condition involving them (and related time-out events) is satisfied and then begins its own execution. At the completion of its execution (or part-way through, as appropriate), tokens are generated and sent to subsequent processes in the data-flow. These tokens may be created and sent to locations as a function of the outcome of the process. Each process executes independently and may exist in any processor within a distributed processing environment.

As a result of the features identified in the previous paragraph, a data-flow naturally permits conditionals, branching, distributed execution, and concurrency. In addition, data-flows can be executed with arbitrary degrees of decentralization as suggested by Figure 9. Figure 9a shows an abstract data-flow involving five processes {A,B,C,D,E}, an outcome-dependent branching at process A, multi-step completion token generation at process B, concurrency of processes C and D, and a logical relation governing the execution of process E. Figures 9b-d show three different implementations with increasing degrees of decentralization. Figure 9b shows no decentralization; all processes are executed by a single multi-tasking processor. *In this case the data-flow scheduling and token-routing is totally contained within the single multi-tasking processor.* Figure 9c shows an environment involving five processors; one processor is used as a centralized controller/scheduler, a second processor executes processes A and E, and the remaining three processors each execute processes B, C, and D. In this case, single start and completion tokens are exchanged between the centralized controller/scheduler and the other process-executing processors as illustrated. *In this case the data-flow scheduling is handled by the centralized controller/scheduler but the actual processes are executed in a distributed environment.* Figure 9d shows another environment involving five processors; in this case each processor uniquely executes one of the five processes {A,B,C,D,E}. There is no centralized control or scheduling. *In this case, the entire data-flow is implemented in a fully decentralized nature.*

The above example is illustrative of the fact that the same rather involved data-flow can be directly supported by a wide variety of environments involving many different types and degrees of decentralization. In addition to this and other useful properties cited above, data-flows also enjoy considerable endorsement outside of the computer literate community, in particular data-flows are useful for specifying corporate, administrative, and financial procedures [6]. As a result, data-flow flavored schemes offer potentials for simplifying configuration specification and user interfaces.



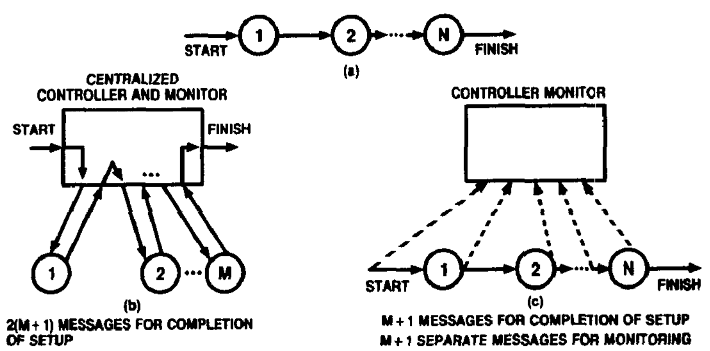
A Data-Flow (a) And Three Implementations With Increasing Degrees Of Decentralization
Figure 9

3.2 Performance-Based Design Considerations

Assuming some decentralization is required in the large-scale reconfigurable systems, interesting observations can be made concerning the role of centralized coordination of a data-flow execution in comparison to decentralized execution.

3.2.1 Execution Consider the simple data-flow shown in Figure 10a. This abstract data-flow contains N processes which execute in sequence. Figure 10b shows a centralized coordination of the execution of this data-flow. Including "start" and "finish" tokens, a total of 2(N+1) messages are required. Note that since the coordination is centralized, any failures of a process execution can be identified and hence recovered from during the execution; also, completion of each process is observed and hence available for monitoring, administration, and accounting (billing) purposes. In particular, completion of the Kth process occurs after 2K messages and is observed after the exchange of 2K+1 messages. Compare this to the decentralized execution shown in Figure 10c. Full execution requires only N+1 messages, half as many as before. Without outside observations (such as shown in the dotted lines) failure recovery and monitoring in an equivalent fashion is not possible, so these messages are now considered. Again N+1 messages are required, so the total number of messages needed is 2(N+1) as in the centralized case. However, the completion of the Kth process occurs after K messages and is observed after K+1 messages, half of that for the centralized case. These and some other remarks to be discussed are summarized in the Table below.

In a decentralized system, message count represents one important degree of complexity. Each message must be created, successfully transmitted/routed/received, analyzed and noted upon. The diversity of the types of messages expands the complexity of the system's elements, and the number of messages


START
FINISH
(a)
CENTRALIZED CONTROLLER AND MONITOR
START
FINISH
(b)
2(M+1) MESSAGES FOR COMPLETION OF SETUP
CONTROLLER MONITOR
START
FINISH
(c)
M+1 MESSAGES FOR COMPLETION OF SETUP
M+1 SEPARATE MESSAGES FOR MONITORING

Comparing Execution Complexities (After Allocations Have Been Made): (a) Abstract Data-Flow, (b) Centralized Control/Monitoring, (c) Decentralized Execution With Centralized Monitoring
Figure 10

scales the loading requirements of the message exchange systems (i.e., the data-communications network linking the system elements.) In most systems each message exchange is likely to introduce delay and potential points of failure, comparing the columns of the table, it is observed that the decentralized approach in many ways makes much more intelligent use of the 2N+2 messages required. The key here is that only half of the messages are required for execution of the data-flow. As a result, the data-flow executes faster* and has fewer points of failure. Since the other half of the messages are used for monitoring, another property is also observed: there is a well-defined distinction between execution-oriented messages and monitoring-oriented messages. As a result, execution-oriented messages can be handled in a separately engineered environment emphasizing speed and reliability. The degree of reliability attained in the execution-oriented environment determines requirements on the monitoring-oriented environment. Since the monitoring information is used for billing, statistical records, and failure-recovery, it is reasonable to design the monitoring-oriented environment for slower speed and lesser reliability. It is also noted that since the monitoring and execution messages are generated in parallel (rather than in sequential interleave), the decentralized approach handles *all* aspects of configuration assembly with half the message exchange time (at the expense of an efficiently doubled message rate). A find observation concerns the fact that the centralized controller used in centralized coordination is required to take N actions even if no failure occurs, while nothing comparable is required is the decentralized case.

The preceding analysis may be extended to more complex (i.e., non-serial) data-flows. To do this requires focus on events *local* to the execution of a single given process as shown in Figure 11. Figure 11a shows a given process receiving m tokens from and transmitting n tokens to other processes in the data-flow. It is assumed that the process shown executes **monolithically**, i.e., begins its execution when received tokens satisfy a logical relation, executes without externally-visible steps, and all n transmitted tokens are then sent. (It is noted that, for the sake of analysis, any non-monolithically executing process may be itself decomposed into a new data-flow consisting of only monolithically executing process.) As shown in Figures 11b and 11c, which respectively illustrate the message flow in the centralized and decentralized approaches, only the received tokens need be considered. (This is because a transmitted token for one process also serves as a received token for exactly one other process.) As shown in Figure 11b, the centralized controller requires as many as m messages to be received for evaluating the logical relation permitting execution; when this relation is satisfied a message is sent to the process signaling that it is to now execute. Execution is confirmed with an additional message. Assume that each of the m+1 messages leading up to the execution of the process encounters a non-zero delay time between the completion of the action generating it and the message's reception; let "T_{max}" be the longest of these times "T_{submax}" be the second longest, and "T_{min}" be the shortest. Then the delay "t" between the time the process in the Figure could execute ideally and in reality must satisfy (assuming no race conditions):

$$(T_{max} + T_{min}) \leq t \leq (T_{max} + T_{submax})$$

Also, note that two messages are required between the action last required for process execution and actual execution.

Compare now the decentralized approach as illustrated in Figure 11c. Here, each of the m received tokens were co-transmitted with monitoring messages as illustrated. Upon execution, the process sends a single monitoring message to the centralized monitor along with the n transmitted tokens sent to subsequent processes in the data-flow. As a result, (assuming no race conditions):

$$T_{min} \leq t \leq T_{max}$$

and only one message is required between the action last required for process execution and actual execution. If all message delays are assumed approximately equal, i.e.,

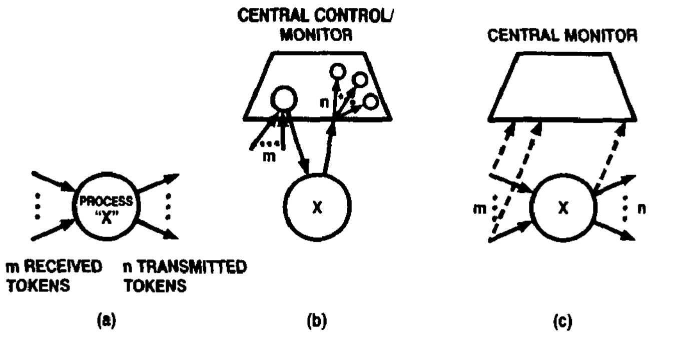
$$T_{min} \approx T_{max}$$

then one sees the decentralized approach emerging with an execution encountering half as much message delay, just as in the serial case.

Comparison of Coordination Methods for Serial Data-flows

Attribute	Centralized	Decentralized with Centralized Monitoring
Total number of messages for complete execution	2N+1	N+1
Total Number of messages for complete monitoring	2N+2	N+1
Number of messages required to executed K^{th} process	2K+1	K
Number of messages required to observe K^{th} execution	2K+2	K+1
Number of actions required by central point during non-failures	N	0

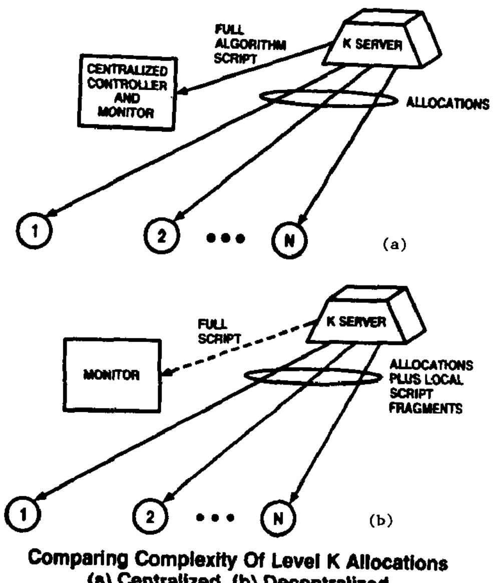
* *Another way to look at this is that for a given user-perceived configuration assembly time, the message network only need operate half as quickly.*



General Data-Flows
(a) Data-Flow Element, (b) Centralized Execution
(c) Decentralized Execution With Centralized Monitoring
Figure 11

3.2.2 Allocations Figure 12 illustrates the differences between the centralized and decentralized approach. As shown each case requires the centralized entity to be presented with the full algorithm script. In the decentralized case, each resource must also be given information about its precise role in the script, hence elongating the allocation messages in size but not increasing the number of messages. Although this elongation is not required in the centralized case, it is the only penalty paid at allocation for decentralization with monitoring.

3.2.3 Comparisons Summarized Decentralization with monitoring elongates allocation messages and increases the instantaneous rate of messages during execution. It does not change the total number of messages involved in execution and monitoring nor does it affect the average message generation rate. However, decentralization does speed up execution, reduce message-related points of failure, and increase the rate at which execute-time failures are detected, typically if not almost always by a factor of 2. In addition, it permits separate engineering of monitoring and execution messages (allowing retribution for increasing the instantaneous message generation rate mentioned above) and prevents additional actions by a centralized entity in non-failure modes.



Comparing Complexity Of Level K Allocations
(a) Centralized, (b) Decentralized
Number Msgs Same; Msgs Slightly Longer Under (b)
Figure 12

4. RESOURCE AVAILABILITY AND CONFIGURATION DISASSEMBLY

It is important to free resources and other types of server allocations when a user has finished with a configuration assembled in their behalf. This is somewhat obvious for otherwise new configuration requests would eventually find no resources available. As it turns out, there are a number of detailed issues closely related to the freeing and general availability of resources. Here some performance and implementation considerations of resource availability and configuration disassembly are discussed.

Note that a given resource, once allocated, cannot be used by another user until somehow made free. A resource is allocated during a configuration assembly and remains unavailable to other users until some time after its use is completed. Thus for a given configuration request, the time a resource is unavailable to other users may be split into three components:

$$T_{unavailable} = T_{assembly.idle} + T_{usage} + T_{disassembly.idle}$$

The time spent in use (T_{usage}) is connected to revenue and cost; it is for these intervals that the resource is provided in the first place. The times spent idle during assembly ($T_{assembly.idle}$) and disassembly ($T_{disassembly.idle}$) phases, however, represent periods when the resource is wasted. The efficiency of the resource allocation system for the duration of a given configuration can thus be characterized as:

$$\text{Efficiency} = \frac{T_{usage}}{T_{assembly.idle} + T_{usage} + T_{disassembly.idle}}$$

The closer this expression is made to 1, the more potential for actual usage can be obtained. To obtain high efficiencies, one requires:

$$T_{usage} \gg T_{assembly.idle} + T_{disassembly.idle}$$

This is particularly important in systems where a great deal of resource sharing is to be expected. Most instances of large-scale reconfigurable systems would fall into this category for sheer reasons of minimizing cost and component-count complexities.

As discussed earlier, it is desirable to allocate resources quickly during configuration assembly for user performance requirements; users typically wish to have configurations made available quickly after a request is made. This force already motivates minimizing $T_{assembly.idle}$. However, one can see now that it is also desirable to minimize the time required to free up resources at the end of their use (i.e., $T_{disassembly.idle}$).

The most straightforward method of freeing resources is to do so on the receipt of a signal from the user that the configuration is no longer needed. In this case each "level.K" configuration function server is notified by some means to free the allocated resource. This, however, can be done in at least two ways:

1. Notification from higher level servers that the configuration is to be disassembled (this is the same way assembly is done);
2. Broadcast notification to each server involved in the configuration from some central point aware of the completion (this requires "look-ahead" operations to provide the centralized point with all the server addresses in advance).

Resources may also be freed using a number of typically context-oriented techniques. In many cases, time-out conditions can be used to supplement user completion messages. In other cases, resources can be implicitly released because of the nature of the allocation discipline requested (for example, a call-forwarding address query, the dump of a file, video frame, or audio segment, etc.). In other cases, resources may be explicitly freed by the user; this may be done by sending a message terminating specific subsets of an otherwise currently active configuration (turning off the video of an audio/video call, etc.).

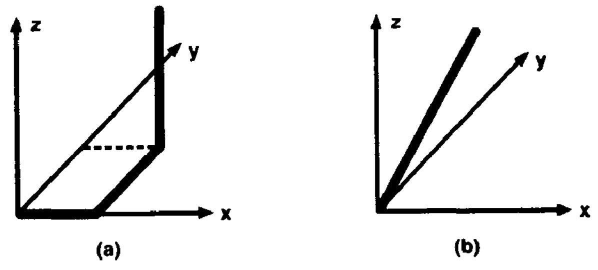
All of the options for resource release may be used in the Threaded/Flow approach. The degree of performance required for resource release is a function of the designed server load, cost of the resources, cost of the release mechanism options, etc.

5. PIPELINING CONFIGURATION USAGE, ASSEMBLY, AND DISASSEMBLY

In many cases it is useful to interleave the assembly, usage, and disassembly phases of a configuration. In some cases this would be done strictly for performance reasons, offering the user a speedier setup and freeing resources as quickly as possible. Another reason would be to provide partial grants of a configuration request while waiting for other resources to become available; the user could be offered key subsets of the configuration (say, audio and graphics features) while waiting to obtain the next available resources for the rest of the configuration (say, video features). Another case would be to provide conditional feature implementation (for example, a user may or may not wish to tie in a graphics editor depending upon the type of image found during a database query whose output is directed to a desktop publishing feature). The latter case is especially useful in functions with very extensive user options and interactive qualities.

Figure 13 shows a way of graphically characterizing such pipelining as multi-dimensional curves. The Figure shows diagrams whose axes are

"x": the number of resource allocations made,
"y": the number of resource-level data-flow steps executed at "level.0",
"z": the number of resource releases made.



Non-pipelined (a) and highly pipelined (b) executions
Figure 13

Assume also that there is only one resource-level data-flow step made for each allocated resource. If each axis is normalized so that the curve is confined to values between 0 and 1, a number of properties, listed below, can be expected. (Variations on the assumptions result in other similar properties.)

1. The curve is non-decreasing in each coordinate (true without normalization);
2. If there are no recursions, the curves must lie within the wedge described by:

$$0 \leq y \leq x$$
$$0 \leq z \leq x$$

3. The shape of the curve determines the degree and style of pipelining. A condition of "no pipelining" is represented by the curve

$\{0 \leq x \leq 1, y=0, z=0\} \cup \{x=1, 0 \leq y \leq 1, z=0\} \cup \{x=1, y=1, 0 \leq z \leq$ (see Figure 13a) while "full pipelining" is asymptotically represented by the curve $x=y=z$ (see Figure 13b). Geometric proximity to these limiting curves gives a graphic intuition for the degree and style of pipelining.

The topic of pipelining also includes two other concepts:

1. "Throw-away" parts of a configuration that are unlikely to be used may be preconfigured anyway in case they are needed quickly. This could be done after the principal part of the configuration is assembled but at some time prior to when the user could first possibly require the rarely used feature.
2. Commonly requested configurations may be pre-assembled to some degree. This is useful when used in an adaptive way, basing the number and types of pre-assembled configurations (as well as perhaps the degree of pre-assembly) on automated statistical observations. (This is comparable to a fast-food restaurant, upon noticing a rush on particular types of hamburgers, precooking a few in anticipation of impending requests.) This type of pipelining is an interesting case since in a way this is pipelining the assembly process with the actual ***request*** process.

6. SERVER DESIGN AND HIERARCHICAL QUEUEING MODELS

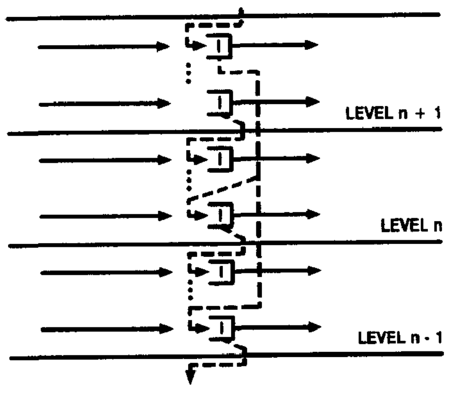
The design of servers and network management systems in the Threaded/Flow approach can at first cut be handled with simple design intuition. This is as done in the design of most computer systems since sophisticated analytic and even simulation techniques often contribute little in practice. However, there is in this case excellent opportunity for the development of some new analytical tools capturing features of the Threaded/Flow approach. In this Section, some design considerations and one promising analytical technique currently under development will be discussed.

In general servers may have to manage resources across a variety of demands. Some requests will be for only brief one-time usage, others for bursty long-term usage, others for long intervals at full utilization. In some cases delay in the allocation will be tolerable, and in other cases it will not. Also, servers will have to administer resources subject to network management conditions. For example, under heavy loading priority may be paid to requests associated with configurations that are almost complete rather than new requests in order to maximize revenue. Expensive resources may be allocated only after guarantees that enough other resources are available to complete the configuration assembly. As a result of these observations, it is seen that the design of servers in the Threaded/Flow system is very interesting and worthy of some theoretical study and dependable design techniques.

The modeling of servers and network management systems in the Threaded/Flow system is nicely handed by ***hierarchical queueing models***. The first model of this type was a two-level model proposed by Schoute to study the creation of tasks resulting from the acceptance of a call-request in a telephone switch controller [7]. Figure 14 shows a more general hierarchical queueing model involving more layers developed by the author [8]. In this model there are separate queues associated with each server within each level. "Customers" may arrive at any level, but customers at a given level can, while being served by their server, ***generate customers for queues at lower levels***. This nicely models many circumstances, such as the scheduling of processes in a batch computer system, allocation of resources within a corporation, or the Threaded/Flow approach presented in this paper. In less obvious ways, the hierarchical queueing model also is useful in modeling resource allocation systems where allocations involve hard or virtual allocations over a range of time-constants. For example, in [8] the allocation of transmission or switch fabric channels to circuit-switched, burst-switched, virtual-circuit, and datagram oriented clients is represented as a two-layer model (a "fast" burst-duration/packet-duration layer plus a "slow" call-duration/connection-duration layer). This is also of great potential use to modeling and designing more general resource servers involved in Threaded/Flow implementations. In [8] the topic is considered in more detail from modeling, analysis, and control design viewpoints. Analytical results appear promising since under standard types of independence assumptions both decomposability [9] and geometric matrix [10] techniques can be applied.

7. APPLICATIONS TO TELECOMMUNICATIONS SERVICE PRIMITIVE SYSTEMS

This discussion is limited in detail due to legal and proprietary issues yet to be resolved. The ideas are fairly straightforward, however. From the service definition viewpoint, this application is based on the following identifications:



General Hierarchical Queuing Model
(A Specific Example Is Shown)
Figure 14

Identification of Threaded/Flow Entities with Service Primitive Entities

Entity	Service Primitive Concept	Threaded/Flow Concept
simpliest resources	resources	"level.0" configuration function (i.e., functional elements)
explicit building blocks	service elements	"level.K" configuration function
functions or services	level.K service	"level.K+1" configuration function ($K \geq 1$)

What has been done here is that the very first level of configuration functions are reserved to create explicit service building blocks called **service elements.** This is done because for most purposes raw resource functions will be too limited to use directly in simple service specifications. In particular, a large-scale network may require finer granularity for its own internal management, operations, and failure-recovery needs than is reasonable to present to users at a user interface. (For example, a resource might be a circuit-switched channel or video laserdisk which, respectively, require routing and linking with specific editors or display controllers before they are of any real use. However, what is viewed as a resource or service element is ***completly arbitrary;*** these examples could just as easily incorporate routing and editors or display control and be viewed as simple resources.)

Other aspects and augmentations include service description languages, service definitions, service description database management, assembly/usage/disassembly languages and interfaces, open architecture issues, call progress, real-time control, billing, failure-recovery, operations, provisioning, service subscription administration, and network management. A few of these are briefly discussed.

7.1 Administration

The operations, billing, call progress, and real-time control functions are implemented by adding specific functions and messages to the various servers. In particular, the highest-level server is used as the centralized point of contact between these functions, users, and the processing of user requests. Communications between servers follows the same hierarchy used in configuration definition. This permits each server to be only locally responsible for affairs it directly delegates.

7.2 Definitions of Services

Services are defined by specifying configurations and passing specific parameters, files, or software to resources. Configurations are specified by software descriptions. These descriptions consists of specific resource request messages and the specific servers to which the requests are to be directed. All information needed to construct a fully decentralized data-flow can be organized in this manner. Because of the Threaded/Flow construct, an entity named in a description can be a resource, an elementary service element, or a full stand-alone service. A given server, however, does not distinguish between these since it only is concerned with affairs within its level and message exchanges with levels immediately below and above. Note each server is responsible for failure detection and recovery for all the servers and resources it explicitly deals with.

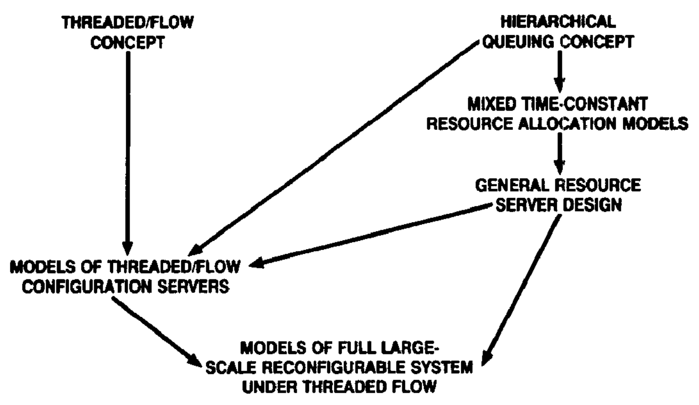
A network defined service is one whose description is provided by a network administered database. A service vendor defined service is one whose description is provided by a service vendor database. A user defined service is one whose description is provided in some manner by the user. User-defined services may be limited to single-layer descriptions of robust services and service elements with significant filtering and security functions; otherwise the integrity of the entire network can be easily compromised.

7.3 Roles for Service Vendors

Service vendors can provide resources, service descriptions, or combinations of both. The interface to a service vendor is expected to be identical to that of a user with the probable exception of support for higher channel capacities.

8. SUGGESTIONS FOR RELATED ANALYTICAL WORK

There is the potential for some interesting formal language work characterizing the intrinsic structure of the Threaded/Flow approach. In addition, category representations [11] using functors are particularly attractive for specifying the synthesis, decomposition, and equivalence of configurations. (It is noted that category theory has seen value of a completely different manner in other work with languages and algorithms; see, for example, [12].) Besides these algebraic studies, there is considerable work that could be done with the notion of hierarchical queueing systems and their control, both from the pure layering and mixed time-constant viewpoints. In addition, there are probably other types of resource allocation and network management models that could be used to study and design the servers used in the Threaded/Flow approach.


THREADED/FLOW CONCEPT
HIERARCHICAL QUEUING CONCEPT
MIXED TIME-CONSTANT RESOURCE ALLOCATION MODELS
GENERAL RESOURCE SERVER DESIGN
MODELS OF THREADED/FLOW CONFIGURATION SERVERS
MODELS OF FULL LARGE-SCALE RECONFIGURABLE SYSTEM UNDER THREADED FLOW

Role Of Hierarchical Queuing Models
Figure 15

9. ACKNOWLEDGEMENTS

The author would like to thank Warren Gifford (Bell Communications Research) and Professor Pravin Varaiya (U. C. Berkeley EECS Department) for their support and encouragement of these ideas over the years. The author is also grateful to Bell Communications Research for supporting the development of the *Integrated Media Architecture Laboratory* (IMAL) [1] where the need for these ideas has provided a wonderful theater for study. In particular, the author wishes to acknowledge the programming design and coding efforts of Chris Bidaut (AGS) and earlier exploratory efforts by Doug Riecken (Bell Communications Research) involved in creating the first working prototype of the Threaded/Flow approach within IMAL. This work has also been nicely complemented by the efforts of summer student Damon Altos (Rutgers University) in creating an extremely sophisticated graphics-oriented monitoring display system for the Threaded/Flow prototype. Finally, but with great thanks, the author acknowledges the valuable editorial review of Laura Pate and Robert Klessig (both of Bell Communications Research).

10. REFERENCES

[1] L. F. Ludwig, D. F. Dunn, "Laboratory for the Emulation and Study of Integrated and Coordinated Media Communication," this SIGCOM conference.

[2] R. G. Loeliger *Threaded-Interpretive Languages* Byte Books, Peterborough, NH, 1981.

[3] M. Broy, ed., *Control Flow and Data Flow,* Springer, New York, 1984.

[4] J. A. Sharp, *Data Flow Computing,* Halsted Press, New York, 1985.

[5] C. R. Vick, C. V. Ramamoorthy, eds., *Handbook of Software Engineering,* Van Nostrand, New York, 1984.

[6] T. Demarco, *Structured Analysis and System Specification,* Prentice Hall, Englewood Cliffs, NJ, 1979.

[7] Schoute, "The Hierarchical Queue: A Model for Definition and Estimation of Processor Loading," *Phillips Technical Report,* SR220-82-3743, October 22, 1982.

[8] L. F. Ludwig, "Hierarchical Queues and the Control of Layered and Mixed Time-Constant Resource Allocations", *to appear.*

[9] Courtis, *Decomposability,* Academic Press, New York, 1977.

[10] M. F. Neuts, *Geometric Matrix Solutions in Stochastic Models,* John Hopkins, Baltimore, 1981.

[11] S. MacLane, *Categories for the Working Mathematician,* Springer, New York, 1971.

[12] P-L Curien, *Categorical Combinators, Sequential Algorithms, and Functional Programming,* Wiley, New York, 1986.