International Journal of Reliability, Quality and Safety Engineering
©World Scientific Publishing Company

THE AMARANTH FRAMEWORK: POLICY-BASED QUALITY OF SERVICE MANAGEMENT FOR HIGH-ASSURANCE COMPUTING*

CAROL L. HOOVER, JEFFERY HANSEN, PHILIP KOOPMAN, and SANDEEP TAMBOLI

Department of Electrical & Computer Engineering and The Institute for Complex Engineered Systems
Carnegie Mellon University, 5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213-1890, USA
{clh, hansen, koopman}+@CMU.EDU, sandeep.tamboli@marconi.com

Abstract: System resource management for high-assurance applications such as the command and control of a battle group is a complex problem. These applications often require guaranteed computing services that must satisfy both hard and soft deadlines. Over time, their resource demands can also vary significantly with bursts of high activity amidst periods of inactivity. A traditional solution has been to dedicate resources to critical application tasks and to share resources among non-critical tasks. With the increasing complexity of high-assurance applications and the need to reduce system costs, dedicating resources is not a satisfactory solution. The Amaranth Project at Carnegie Mellon is researching and developing a framework for allocating shared resources to support multiple quality of service (QoS) dimensions and to provide probabilistic assurances of service. This paper is an overview of the Amaranth framework for policy-based QoS management, the current results from applying the framework, and the future research directions for the Amaranth project.

Keywords: quality of service (QoS) management, resource management, probabilistic assurances of service, system management for high-assurance computing.

1. Introduction

High-assurance applications such as the command and control of a naval battle group require the timely allocation of resources to enable critical computing on demand. The allocation of resources to support the various mission activities of a battle group is challenging because the necessary processing and data communications of multiple surface ships and aircraft are sporadic with periods of inactivity and bursts of activity. Traditionally, system designers dedicate resources to the highly critical tasks and share other resources among less critical tasks. This solution satisfies high-assurance demands at the cost of potentially inefficient use of resources, a situation that can result in unnecessarily high space and maintenance requirements. In addition, the integration of secure data and communications among tasks operating on heterogeneous, distributed platforms can be complex. Inefficient use of resources multiplies the problem.

A more cost efficient solution would be to share a set of heterogeneous resources among the various distributed application and system tasks. Using standardized communication protocols would simplify the design of tasks that can reliably communicate across distributed computing nodes. In particular, it is necessary to multiplex the network bandwidth among soft versus hard real-time tasks and across bursty versus steady-stream communications. System designers seek to configure a set of resources that have the potential to maximize the utility of application tasks while minimizing system costs.

*The Amaranth Project at Carnegie Mellon is supported by the Quorum Program of the Defense Advanced Research Projects Agency (DARPA) under agreement number N66001-97-C-8527. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

Likewise, application designers want critical tasks to operate with high assurance and less critical tasks to execute at performance levels that match the needs of the intended users. User preferences can vary across QoS features such as timeliness, dependability, security, and application-specific performance. What is needed is an approach for managing resources across multiple quality of service (QoS) dimensions in a way that maximizes their value across user requests while providing some known level of guaranteed service in the event of high resource demands or resource failure.

The Amaranth solution is a user-centric framework for managing resource allocations along multiple QoS dimensions and viewpoints. The paper overviews the Amaranth framework as well as the analytical techniques to support policy-based QoS management with probabilistic guarantees. The remainder of the paper is organized as follows.

- *Section 2* characterizes the target application as well as the representative application used to validate the Amaranth approach to QoS management.
- *Section 3* contains a discussion of related research.
- *Section 4* defines the Amaranth terminology and functionality.
- *Section 5* details the components of the Amaranth QoS management system architecture.
- *Section 6* describes a language, called Q, for specifying Amaranth QoS parameters.
- *Section 7* discusses utility-based QoS management.
- *Section 8* describes QoS contracts and probabilistic guarantees.
- *Section 9* presents a reserve capacity approach to admission control.
- *Section 10* draws conclusions about the current research results.
- *Section 11* suggests future research for the Amaranth project.

2. Target Application

The goal of the Amaranth research is to develop capabilities to support the underlying distributed communication and computation infrastructure that will become increasingly important in future commercial and military systems. The military units of the twenty-first century will be radically different from their predecessors. Their computer systems will support various types of computing from parallel processing embedded in the twenty-first century surface-combatant (SC-21) computing platform to distributed processing embedded in tanks such as the M2 A1 Bradley which is equipped with a missile subsystem. These systems will run a collection of software agents embedded in a communication and computing infrastructure that provides cognitive support for the interconnection of soldiers, sailors, and commanders through the "tactical internet" of the digital battlefield. As another example, the "soldier phone" applies wireless and multimedia communications to the accomplishment of battlefield objectives.[1]

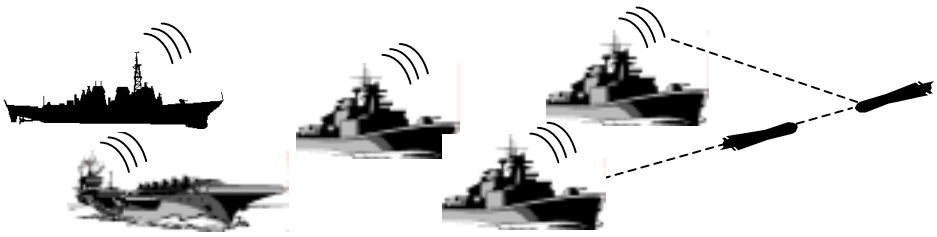

Fig. 1. Battle group communication among ships with missile tracking and interception.

As shown in Fig. 1, a typical configuration would be a group of surface ships with a computing environment that consists of a set of heterogeneous computing nodes. These nodes execute software to support mission-critical functions such as planning, communication, and missile tracking and deployment, as well as "housekeeping" operations such as the management of inventory and personnel records. The bandwidth for a ship's local network would be high, while the bandwidth among ships would be limited.

The mix of tasks utilizing bandwidth would consist of both multimedia data streams and short, intermittent weapons control messages. Both types of traffic may experience long periods of relative inactivity (e.g. minutes to hours) with short, intense periods of high activity (e.g. milliseconds or seconds). Though the bandwidth for the active periods of the multimedia stream would most likely be greater than that for the bursty weapons control communication, the control messages would be more critical to the mission of the battle group and also less resilient to degradations in QoS. Many multimedia applications can perform effectively with reduced rates of frame update and can employ compression techniques to minimize bandwidth, whereas weapon control messages are time-critical.

System designers cannot afford to size their systems for the worst case expected application mix and behavior to achieve maximum QoS for all task requests. Since many applications can function usefully at a degraded but acceptable QoS for short periods of time, system designers can reduce the system size accordingly. The "right" size system can service critical tasks during periods of peak activity while providing less critical tasks with minimal periods of degraded service during normal operations. The system resource management policies and mechanisms should be able to adapt to variation in workload due to sudden bursts of new jobs such as incoming mission-critical alarms. They should also be able to provide a high degree of assurance that critical tasks will function properly despite fluctuations in equipment availability due to maintenance downtime and battle damage.

The Amaranth framework is a resource management strategy to enable system designers to scale their systems to minimize cost and to maximize the utility of the system to the anticipated set of users. Replication of the exact application mix of the battle group scenario described in the previous paragraphs is not feasible, so representative workloads simulate the target application mix in our testbed. The representative application mix consists of a video conferencing application with periods of varying activity along with workload generators to simulate the burstiness of critical missile activity. Though our approach for maximizing utility scales to multiple resources, our current focus is on managing the allocation of network bandwidth. Current policies implemented within the Amaranth framework enable baseline admission of all task requests with 100% assurance. With proper sizing of the total available network bandwidth, all tasks of the target scenario are admissible with a base level of service. Some tasks may receive higher levels of service depending on the system load and the QoS policy active within the Amaranth system.

3. Related Research

Earlier work focused on managing QoS across the layers of the network protocol and on defining the meaning of QoS from source to destination nodes (end-to-end QoS). The advent of multimedia workstations and high-speed networks enabled a new class of applications demanding continuous network bandwidth to support streams of data. Critical distributed control applications requiring reliability and guaranteed bounds on message

latency, along with multimedia, demonstrated the inadequacy of best-effort communications and thereby spurred research in the area of QoS management of network resources. Campbell et al. formally defined the concepts of flow and flow management. They proposed a QoS-A architecture that is a layered architecture of services and mechanisms for QoS management and control of continuous media flow in multi-service networks.[2] They argued that meeting QoS guarantees in distributed multimedia systems is fundamentally an application-to-application (end-to-end) issue. The architectures that they reviewed focus on QoS in the context of individual layers of a network protocol.[3]

QoS management currently resides primarily in the policies and mechanisms to route packets of data. Noteworthy is Nahrstedt and Smith's QoS Broker model for specifying application requirements and translating these requirements into negotiated resource allocations. The QoS Broker acts as an intermediary between application processes and the OS/ network protocol subsystem to communicate the application needs to the lower level services. The broker orchestrates network resources at the source and destination nodes by coordinating resource management across the communication layer boundaries within an end-point node.[4]

Researchers in the EPIQ project at the University of Illinois at Urbana Champaign designed an open run-time environment for hard real-time applications. The environment consists of a distributed QoS management architecture and middleware that accommodates and manages different dimensions and measures of QoS. The middleware supports the specification, maintenance, and adaptation of end-to-end QoS. This work integrates real-time scheduling algorithms within a QoS management framework.[5, 6]

Abdelzaher and Shin designed a middleware layer, called qContracts, which is a programming abstraction for building performance-assured services. Programming with qContracts allows creating, manipulating and terminating QoS contracts with clients or client categories to achieve performance guarantees. The middleware enforces the contracted QoS on behalf of the service programmer. This approach provides portable mechanisms for enforcing contracts; but the middleware does not have control of the operating system resource allocation to enable hard real-time guarantees. On the other hand, the middleware can approximate QoS guarantees that are suitable for a large class of soft real-time applications, such as web services and e-commerce.[7]

Current research also delineates QoS from multiple viewpoints such as user or application, system, and resource.[8] Sabata et al. outlined a taxonomy that classifies QoS parameters from different viewpoints. They specified QoS as a combination of metrics and policies. Metrics measure specific quantifiable attributes of the system components, and policies dictate the behavior of the system components.[9] Researchers at SRI outlined issues in managing resources for complex distributed systems. They formulated the Logical Application Stream Model (LASM) for capturing a distributed application's structure, resource requirements, and relevant end-to-end QoS parameters. They also developed the Benefit Function (BF) model for expressing user QoS preferences and for gracefully degrading an application's QoS under certain conditions.[10, 11]

More recently, Bashandy et al. developed a protocol architecture to define distributed multimedia systems from the application as well as the service providers' points of view. Formally defined as finite state machines, their architecture consists of an application layer, a configuration and synchronization layer, network layers, as well as a database and computation backbone. Each layer consists of independent functional units that communicate through a standard framework of messages.[12]

Chen and Hsi formulated the design of admission control algorithms for real-time multimedia servers as a reward optimization problem with the "reward" referring to the value which the system receives after servicing prioritized clients based on the QoS requested and delivered. They classified admission control algorithms as "deterministic" or "best-effort" regarding quality of service (QoS) control and as "priority-reservation" or "no priority-reservation" regarding reservation control.[13] Likewise, Amaranth admission control attempts to maximize the "reward" to the system, but the formulation of this reward depends on the active QoS policy or policies. In the Amaranth context, reservation implies the holding of reserve resources in anticipation of random and bursty demands for service. Amaranth is one of the first QoS management systems that we have seen to apply traditional feedback control theory to determine appropriate resource allocations that maintain a "set-point" of reserve capacity.

The Amaranth research differs from other work by emphasizing the following features.

- QoS management across multiple QoS dimensions defined with respect to the needs of the client applications.
- QoS contracts with probabilistic assurances that users will receive contracted resources.
- Fault monitoring to detect and predict resource failures in order to help prevent QoS contract violations due to resource downtime.
- Monitoring of resource usage patterns and reserve capacity to preserve contracted resource allocations in the event of bursts of task requests.

Unlike other research that focuses on mechanism across multiple system layers for managing QoS, Amaranth research focuses on policy development and deployment. The goal of the Amaranth framework is to provide a flexible environment in which system designers can deploy their own QoS management policies.

4. Amaranth Terminology and Functionality

This section presents terminology to describe the Amaranth framework and to develop mechanisms for communication between applications and the Amaranth QoS management system. A use case diagram describes interactions between users and Amaranth.

QoS Dimension:

User point of view - A QoS dimension is a domain-specific feature or application requirement whose performance level can be observed or controlled. These features may be functional such as application quality (e.g. frames per second or resolution for a multimedia application) or type of security (e.g. encryption). They may relate to execution behavior such as timeliness (e.g. latency and allowed lateness) or dependability (e.g. availability and reliability). Our model for QoS is an n-dimensional space with utility (reward) values and resource requirements associated with each point in space.

System point of view - A QoS dimension helps to define the QoS space in which each point corresponds to a specific allocation of required resources and system services.

Application:

User point of view - An application is a type of software that can execute at differing levels of performance. The application designer specifies the available levels of service for each QoS dimension. The application user specifies the mapping between the points in the QoS space and their associated utility values. Higher utility values signify greater degrees of usefulness or importance of the related points in the QoS space.

System point of view - An application is a class which, when instantiated and executed, yields utility in return for consumed resources. The application designer with the help of the system determines the required resources to achieve each point in the QoS space. The system has available to it a specification of (or a function to determine) the utility value versus resource consumption for each point in the QoS space.

Session:

User point of view - A session is an instantiation of an application with a contracted probability of assurance that the system will provide resources sufficient to achieve an agreed-upon level of service for the contracted interval of time. The user associates the usefulness of the session with the contracted QoS level.

System point of view - A session is an entity to which system resources have been committed for a fixed interval of time with a contracted probability of assurance that the resource allocation will not be reduced during the interval.

Session Request:

User point of view - A session request is a request to negotiate a QoS contract and to start an application.

System point of view - A session request involves the negotiation of a QoS contract and a decision of whether or not a contract for the requested level of service and duration can be provided with the requested probability of assurance.

Contract:

User point of view - A contract is a guarantee that within a specific session the system will provide the necessary resources to enable the associated application to perform at the contracted QoS level with the contracted probability of assurance.

System point of view - A contract is a commitment to allocate the resources necessary to enable the application to perform at the contracted QoS level with the contracted probability of assurance.

QoS Violation:

User point of view – A QoS violation is a state in which the application cannot execute at the contracted QoS level because the system has not provided the contracted service.

System point of view - A QoS violation is a state in which the system has over-committed resources and thus cannot support the contracted QoS level of a session.

QoS Availability:

User point of view – QoS availability is the fraction of time over the duration of a session for which the application experienced no QoS violations.

System point of view – QoS availability is the fraction of time over the duration of a session for which the system actions resulted in no QoS violations.[a]

QoS Reliability:

User point of view – QoS reliability is the probability $R_{\{Qs\}}(t)$ that the application has experienced no periods of QoS violation during the time interval t beginning with the start of session s.

System point of view – QoS reliability is the probability $R_{\{Qs\}}(t)$ that the system actions resulted in no QoS violations during the time interval t starting with the beginning of the session s.

[a] Lowered resource allocations due to resource shortages could result in a QoS violation.

QoS Policy:

User point of view - A QoS policy is a course of action by the system that affects the negotiation of contracted QoS with probabilistic assurances and fulfillment of the resulting contract.

System point of view - A QoS policy is a method of action that determines the outcome of the contract negotiation for a session request and guides the way in which the system fulfills the contract. The system seeks to allocate resources to satisfy system goals such as maximizing utility while admitting all application requests at a baseline or better QoS level.

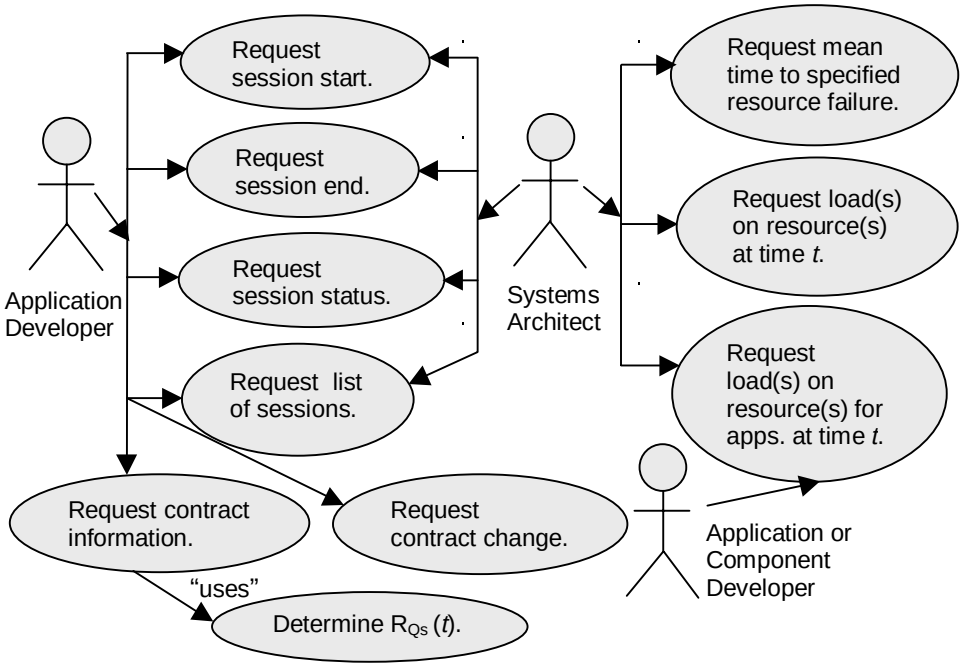

Fig. 2. Amaranth use case diagram.

As shown in Fig. 2, software application or component developers, application users or clients, and system architects would interact with Amaranth. Each of these users would specify the QoS parameters that relate to their roles. As discussed below, these users would specify the QoS parameter values necessary to contract a specified level of QoS (point in the QoS space) and level of assurance for the duration of a session.

The *Application* or *Component Developer* would specify application performance levels or QoS points for relevant QoS dimensions such as application-specific, timing, dependability, and security requirements. The following are examples of application-specific and timing requirements.

- Mathematical model of task arrivals.
- Acceptable time delay between task arrivals and completions (used to calculate deadlines).

- Tolerance to lateness or a function that specifies how the value of the task degrades with respect to the time by which a deadline is missed.
- Mapping between the QoS space and resource consumption.
- Probabilistic model of resource consumption for a particular application.

The *Application User* or *Client* would provide the mapping between the QoS space and the utility or "desirability" of each point in the space. The *System Architect* would specify the weighting or priority of each application or user registered with the system and the system size needed to guarantee minimum acceptable QoS for all critical applications in the event of the worst case load of application requests.

The UML use case model in Fig. 2 shows a user's interaction with an Amaranth system. For demonstration purposes, the user is a human. In practice, the user may be an application program contracting with the Amaranth system for a specified level of resource allocation. The following list indicates the major functions of the Amaranth QoS management system.

1. To receive, process, and admit/reject session requests.
2. To contract, for each session request, levels of QoS and assurance that achieve system goals as specified by the active QoS policies. Maximizing system utility is an example system goal. The corresponding policy would be to allocate resources to tasks so that the utility across all tasks is maximal.
3. To allocate resources according to session contracts.
4. To monitor system parameters such as the following elements.
   - resource usage across sessions
   - current resource usage per session
   - resource usage per session over time
   - system resource capacity instantaneously and over time
5. To adjust the resource allocations when necessary to satisfy the session contracts while adhering to active system policies and reacting in a timely manner to resource failures and other system disturbances.
6. To monitor and notify the session planner of pending, likely, or actual resource failures and expected downtimes.
7. To enable the human user to enter session requests and monitor session behavior through visual and tactile interaction with the system.
8. To enable the human user to simulate session requests without the use of the actual resources.

The next section diagrams and discusses the Amaranth system architecture and the component interactions to service a request to run an application within a session of contracted resources.

5. Amaranth Architecture

The Amaranth architecture consists of the components shown in Fig. 3. The description of how each component contributes to the Amaranth QoS management appears in Subsection 5.1. The policy stack component embodies the policy modules that are currently installed in the Amaranth system testbed. The system administrator or researcher activates the installed policy modules that the Amaranth QoS management system will use to determine appropriate resource allocations to application requests. The Amaranth framework currently focuses on utility-based resource allocations with probabilistic guarantees and resource reserves. The policy stack feature enables the

framework to be extended to support future policy development efforts. Subsection 5.2 describes the user interface tools to direct a Sesco agent.

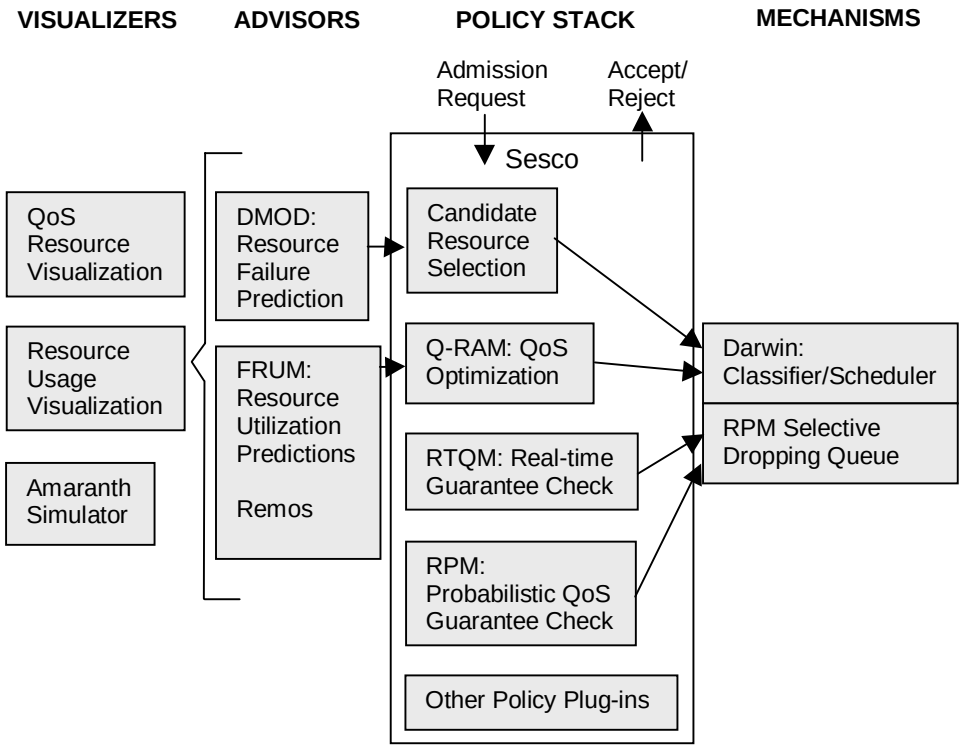

Fig. 3. Amaranth Architecture.

5.1 Amaranth system components and their functions

The following components compose the Amaranth system architecture.

- QoS/Resource and Resource Usage Visualizers - Assist users in making QoS/ resource trade-offs. The network visualizer currently displays the routes utilized by one or more application sessions and enables the user to examine how they interact. In the future, this visualizer will use dynamic aggregation techniques to simplify information about large networks of more than 100 nodes into manageable maps.
- Amaranth Simulator - Given the application requests for service, available resources, and active policies, simulates the resource allocations that would be made by the Amaranth QoS management system.
- DMOD (Dependability Module) - Uses historical information about past computing node failures to estimate resource availability and to predict node failures.
- FRUM (Forecasted Resource Usage Module) - Uses historical trends to predict future resource usage.

- Remos (Resource Monitoring for Network-Aware Applications) - Enables network-aware applications to obtain relevant information about their execution environment.[14] Via the Remos API, FRUM can determine the current loads on network links being monitored by the FRUM module.
- Candidate Resource Selector - Determines the appropriate resources that are available for a particular application request.
- Q-RAM (QoS Resource Allocation Model) - Determines resource allocations that optimize total system utility across all application requests for resources while satisfying resource constraints.[15-18] Section 7 presents an explanation of utility-based QoS management.
- RTQM (Real-Time Queueing Modeler) - Uses arrival and servicing models as well as real-time queueing theory to estimate the distribution of tasks that would be late due to system resource constraints.[19-22] Further discussion of real-time queueing appears later in this section.
- RPM (Resource Priority Multiplexing) - Uses resource usage profiles to validate assurance levels and compute scheduling parameters. Section 8 is a discussion of contracts with probabilistic guarantees.
- Sesco (Session Coordinator) - Receives and processes session requests, queries the monitoring modules DMOD and FRUM as well as the active policy advisor modules (e.g. Q-RAM, RPM, or RTQM) to make resource allocation decisions, and controls low-level mechanisms to implement resource allocations.
- Darwin Scheduler - Serves as a resource management mechanism for application-specific handling of network traffic and sharing of resources among traffic streams that cooperate.[23] Amaranth uses Darwin to reserve network resources and to enforce bandwidth allocations.
- RPM Scheduler (Resource Priority Multiplexing Scheduling Mechanism) - Implements resource priority multiplexing via a selective dropping queue mechanism.

Embodied in the components Q-RAM and RPM, the current system policies are to maximize system utility across application requests and to validate the probabilistic assurance that a given session will receive the resources for which it contracts. System utility is a weighted measure of the user-defined utilities associated with the levels of service (points in the QoS space) of the active applications and of the new requests. RPM analyzes the resources allocated across sessions and uses their probabilistic resource requirements to determine the probability that each session will receive its contracted resources. Using arrival, servicing, and resource availability models, RTQM provides Sesco information about the expected distribution of application tasks that will not be able to meet their deadlines.

Timeliness is an important quality dimension for the types of applications targeted for Amaranth QoS management. For example, the system may need to provide support for real-time control systems in which the control loops have hard timing constraints as well as for voice and video packets that have constraints on latency. Real-time resource management is fairly well understood for the classical periodic task model, but Amaranth must handle tasks with more stochastic behavior and must strive to achieve high levels of resource utilization while providing as high a QoS utility as possible to all the applications. The stochastic behavior of tasks makes it difficult to ensure that the system will behave properly with respect to real-time requirements. A new approach to this problem, called real-time queueing theory, provides a set of analysis techniques which

offer the ability to combine the system predictability associated with hard real-time scheduling theory and the generality of stochastic task behaviors associated with queueing models.[19-22] The RTQM component implements this theory.

In an Amaranth-managed system, FRUM and DMOD execute on each computing node to track usage and failure patterns. Sesco executes along with its policy advisors (currently Q-RAM, RPM, and RTQM) on the network routing nodes. The Darwin Scheduler executes on the routing nodes to enforce network reservations.[23] The visualizers run on the user nodes, and workload generators execute on nodes which system experimenters are using to perturb the system. The testbed in the Amaranth laboratory consists of interior routing nodes running FreeBSD (a version of Unix) and exterior user nodes executing Microsoft's Windows NT. The components are implemented as C++ modules.

Fig. 3 also illustrates the component interactions that enable the session coordinator, Sesco, to process a session request. Sesco is responsible for receiving and processing session requests to achieve the goals of the active system policies. The policy stack keeps track of the active policy modules and helps SesCo to integrate the system policies. Each Sesco agent controls the use of the network links connected to the routing node on which the agent executes. In the event that a requesting application would like to communicate with another application running on a node outside the set of nodes managed locally, the local Sesco calls a global Sesco Coordinator to communicate with the Sesco agent that manages the network in which the target node is located. There can exist a hierarchy of Sesco agents and Coordinators, each of which is assigned a domain name.

5.2 User interface tools for directing a Sesco agent

There are three user interface tools that can be used to direct a Sesco agent: *arq*, *multiarq* and *sesstat*. Implemented in Tcl/Tk[24], these tools communicate with Sesco via a TCP connection.

Arq enables the user to manually make admission requests for legacy applications that cannot directly communicate with Sesco or to modify the default utility curves when making requests through QoS-aware applications. The interface consists of the following pages and features. Section 6 contains a description of the Q specifications referenced here.

- Login page to specify a Sesco user name and password.
- Application selection page to specify an application, a contract, and a mode.
- Application parameters page which is automatically configured from the "**ask**" declarations in a Q specification.
- Utility curve editing page to modify the default utility curves specified in the contract part of a Q specification.
- Notification page to display the actual QoS setpoint selected by Sesco.
- Next and Previous buttons to navigate among the pages.

Multiarq, a tool used for testing Sesco, can generate a stream of simulated admission request and session termination requests with specified distributions. While there are no real sessions running, the interaction with Sesco is the same as with a real application. The simulation model is that of a fixed pool of sessions, each of which can be either in an active or dormant state. When a session changes from dormant to active, *multiarq* sends Sesco an admission request. When the session changes back to dormant, it sends a session termination request to Sesco. A configuration file, which contains a list of session specifications, directs the simulation. Each session specification contains the information

that would be entered through the *Arq* tool interface as well as average times between dormant to active transitions and between active to dormant transitions. The assumption is that both transition times are exponentially distributed.

Sesstat is the Sesco administrative tool to configure the system, to set policy, and to display system performance information. *Sesstat* has the following pages that are accessed through an index tab interface.

- *Map page* to display a network map of nodes managed by the Sesco agent. Clicking on a node or link will display detailed information about the link. Clicking on a session name from the session list displayed on this page will show the resources used by that session.
- *Session page* to display detailed QoS information about a session. Clicking on a session name from the session list will display the application, contract, user, and QoS setpoint information for that session. A user with Sesco administrative privileges can forcibly change the QoS setpoint.
- *QoS page* to display graphs of global QoS performance information such as the global utility value, the number of sessions in the system, and the QoS renegotiation rate.
- *Users page* to edit the user database and to change user weight factors. Only user with Sesco administrative privileges can access this page.
- *Policy page* to configure policy information. The display consists of a graphical representation of the current policy stack and advisors similar to the stack shown in Fig. 3. Interface commands enable the user to load and unload policies and advisors, to change the order of policies in the policy stack, to register policies as being clients of an advisor, and to set the configuration parameters of policies and advisors.
- *Log page* to display the Sesco log file that contains records of events such as session admissions and terminations. A log level filter can be used to display only events at a certain status level and higher. Only users with Sesco administrative privileges can access this page.

Section 6 briefly overviews the Q language used to specify Amaranth QoS parameters.

6. Specification of QoS Parameters

Amaranth uses a QoS specification language, called "Q." Based on the Q-RAM paradigm, Q describes the QoS requirements of an application in terms of an application profile that defines the mapping between QoS and resource requirements and with respect to a user profile that defines the mapping between QoS and utility. Unlike other QoS contract languages[25], Q specifications indicate QoS preferences rather than specific levels of service or procedural definitions of QoS responses to changes in resource availability. There are three main types of blocks in Q: system, application, and contract.

6.1 System declarations

System blocks describe system-level state information such as the available resources, network configuration, and user database. A system administrator can specify the network configuration and resource availability information manually, or a resource discovery system such as Remos[14] can generate the specification automatically.

The user database contains information about a set of users, the users' passwords, and a group affiliation. A group definition is composed of a set of mode definitions corresponding to different situations. Associated with each mode is a *weight factor* that

acts as a modifier to the user profiles and a *log-file prominence level* that determines the degree of prominence for logging of session requests made under a particular mode. For day-to-day use, a user might use "Normal" mode with a modest weight factor and a low log-file prominence. In an emergency situation, a user can use "Priority" mode to get a higher weight factor and thus more resources; but the higher log level will cause such session requests to be logged prominently in the log file to discourage abuse.

System blocks have the form:

```
system [<name>] {
  [declarations...]
};
```

where "name" is an optional identifier to denote a particular Sesco manager. Sesco uses only system blocks without a name or those with names that match the domain name of the Sesco agent reading the Q system file. This allows files without a name identifier to be shared among multiple domains and, thereby, reduces the maintenance burden.

6.2 Application declarations

Application blocks define the QoS requirements of an application as a function of QoS. An application block has the syntax:

```
application <name> {
  [declarations...]
};
```

where "name" is the name of the application. The first part of an application block typically defines the QoS dimensions for the application as well as possible settings for each of the dimensions. For example, a video conferencing application such as *vic*[26] might have the following declarations:

```
qos frame_rate {5, 10, 15, 20, 25, 30};
qos quality_factor {15, 11, 7, 3};
qos security {none, des, des3};
```

to define frames-per-second, quality factor, and security level as QoS dimensions. The values in the braces are possible QoS levels for each of those dimensions.

An optional annotation to a literal or symbol in a Q specification file consists of the # operator followed by a string. Sesco uses such annotations to automatically configure dialog boxes in the user interface. For example, an annotated version of a quality factor declaration might appear as follows.

```
qos quality_factor#"QualityFactor:"{15#"low",
  11#"medium",7#"high",3#"very high"};
```

Such strings are particularly useful when the actual symbol values are hard to interpret by the user. In this example, the quality factor values are those for an *H.261*[27] video stream in *vic* where lower values mean better quality. Additional operators allow one to associate icons that are to be used in the user interface.

The second type of declaration in an application block is an "**ask**" statement. This type of statement defines auxiliary parameters needed to complete an admission request. This information can come directly from a QoS aware application or from a user interface to enable manual admission requests for legacy applications. For the *vic* example, the "**ask**" block might be as follows.

```
ask {
  host src#"Source Host:"="algol.ices.cmu.edu";
  host dst#"Destination Host:"="doradus.ices.cmu.edu";
  int vsrc#"Source Port:"=5555;
  int vdsr#"Destination Port:"=5556;
}
```

Each declaration in an **ask** sub-block defines one parameter. For example, "**src**" is the parameter for source host of the video conference. As with the QoS dimension declarations, the "**#**" syntax defines a string to be used in user interfaces for manual admission requests. The values on the right specify default values for use by a user interface, but QoS-aware applications may override these values. The typed declarations (e.g. "**host**", "**int**", etc.) allow type checking before an application request is actually presented to the policy stack. The values entered via a user interface or supplied by an application are bound to the variable names given in the declaration.

The third type of declaration is a "**flow**" definition. For example, the following Q specification defines a flow called "**videoflow**" that *starts* at the host specified by the "**src**" variable on the port specified by the "**vsrc**" variable and that *ends* at the host specified by the "**dst**" variable on the port specified by the "**vdst**" variable. An optional body to the **flow** declaration can specify additional properties of the flow.

```
flow videoflow $src/$vsrc -> $dst/$vdst {
  protocol=udp;
}
```

After the **flow** declarations come the abstract resource definitions. These define abstract resources used by the application that will be mapped to physical resources by the RSEL policy in the Amaranth policy stack. In the video conference example, one might write the following Q specification to describe resources needed for the flow and CPU at the source and destination nodes and associate them with the symbols "**Rvpth**", "**Rscpu**", and "**Rdcpu**."

```
resource Rvpth = <<bandwidth videoflow>>;
resource Rscpu = <<cpu $src>>;
resource Rdcpu = <<cpu $dst>>;
```

The most important statements in the application block are the resource requirement declarations. Specifications of resource requirements include one or more "**map**" declarations that are combined using a "**compose**" statement. Each **map** declaration maps zero or more QoS dimensions to a vector of symbolic values. For example, the Q specification below defines a **map** named "**FQ**", which maps each combination of settings for the QoS dimensions "**fps**" (frame rate) and "**qual**" (quality factor) to the three

symbolic values "**FQ.bw**", "**FQ.scpu**", and "**FQ.dcpu**". For example, when **fps** is "15" and **qual** is "1", **FQ.bw**, **FQ.scpu**, and **FQ.dcpu** receive the values 1700, 240 and 240 respectively.

```
map FQ [fps, qual] -> [bw, scpu, dcpu] {
  <5,1>:   <1100, 163, 163>;
  <10,1>:  <2000, 293, 293>;
  <15,1>:  <1700, 240, 240>;
  <20,1>:  <2150, 286, 286>;
  <25,1>:  <1900, 273, 273>;
  <30,1>:  <2100, 300, 300>;
  ...
}
```

An **application** block may contain one or more **map** declarations, and each QoS dimension may appear in one or more **map** declarations. Via "**compose resource**" statements, the mappings appearing in each of the declarations operate in combination. For example, the CPU requirements on the source node might be as given below.

```
compose resource Rscpu = FQ.scpu + SEC.cpuBase +
  FQ.bw*SEC.cpuScale;
```

The CPU requirement is the sum of the CPU requirement for the video encoding, a base CPU required by the encryptor, and the bandwidth of the video flow multiplied by a CPU scale factor determined by the encryption setting. The CPU requirement for the video encoding is derivable from the frame rate and quality factor, both of which are specified in the **FQ** map. In the special case where symbols specified in the right-hand side of each of the **map** blocks match one of the declared abstract resources and no **compose resource** statement is given, the resource requirements are the sum of the mappings in each of the blocks.

By supporting multiple **map** blocks instead of a single **map** block, it is possible to define and empirically estimate the resource requirements even for large numbers of QoS dimensions and settings on each dimension. For example in an application with ten QoS dimensions and ten settings on each dimension, there would be a total of 10^{10} set-points. It would be impractical to empirically measure and specify this many set-points with a single **map** block, but the combination of **map** blocks and **compose** statements makes it possible to characterize the resource requirements for applications with large QoS spaces.

6.3 Contract declarations

The final block type is the contract declaration. Contract declarations specify the utility curves for a user. Contract blocks also use the `application` keyword but contain a two-part literal specifying the base application name and the contract name. For example, the specification shown below is an excerpt from the **default** contract for the application vic.

```
application vic::default {
  utility A (1.0) [fps] {
     <5>:  0.24286;
```

```
        <10>: 0.44844;
        <15>: 0.62246;
        <20>: 0.76976;
        <25>: 0.89455;
        <30>: 1.0;
    }
    ...
}
```

The most important types of declarations in contract blocks are the utility curve definitions. A utility curve declaration can specify a utility curve for a single QoS dimension or a joint utility curve for multiple QoS dimensions. The declaration also specifies a weight factor for the dimension. The above example specifies the utility values for the QoS dimension of frames per second (*fps*). The feasible *fps* levels for application *vic* range from 5 to 30 fps, and the utilities associated with these levels range from 0.24286 to 1.0. The weight associated with *fps* is 1.0. As with the **`map`** declaration, it is possible to specify a "**`compose utility`**" statement; but it is more common to simply use the default sum of the utility mappings.

While contract blocks typically include only **`utility`** declarations, any of the declarations that can be specified in the application block are allowable in the contract block. When an application admission request is made, the client specifies an application name and a contract name. The system combines the matching application and contract blocks into a single composite block that is the evaluated by the parser.

Section 7 explains the utility-based resource management policy embodied in Q-RAM.

7. Utility-Based Resource Management

The goal of the Amaranth QoS management is to allocate the system resources in such a way that user applications can operate effectively according to each application's requirement for assured resource allocations without having to under-utilize or dedicate resources. This goal motivates the Q-RAM policy: the maximization of system utility with respect to application utility across multiple performance levels for each QoS dimension. The feasible combinations of performance levels across the relevant QoS dimensions form an n-dimensional space of QoS points where *n* is the number of QoS dimensions. The user can specify "desirability" (utility) values for individual QoS points or a function that will assign utility values to a domain of points.

The Amaranth advisor Q-RAM determines a near optimal allocation of resources to maximize the system utility without exceeding the available resources. The system allocates to each admitted application task an amount of resources that is greater than or equal to the baseline or minimal amount needed by the task. The Q-RAM advisor provides polynomial-time algorithms to determine resource allocations that achieve near optimal utility for systems consisting of a single resource and multiple independent QoS dimensions or multiple independent resources and a single QoS dimension.[15-17]

The formulation by Lee et al. illustrates the complexity of maximizing utility for multiple resources and multiple QoS dimensions, the MRMD optimization problem. MRMD is an NP-Hard problem.

Let K_{i1} ,..., $K_{i\,|Qi|}$ be an enumeration of the quality space, Q_i, for task T_i.

Let ρ_{ij1} ,..., ρ_{ijNij} be an enumeration of the resource usage choices (trade-offs among

different resources) associated with K_{ij} for T_i, where N_{ij} is the number of such resource usage choices. In particular, we require $\rho_{ijk} \models_i K_{ij}$. The resource vector ρ_{ijk} must provide QoS levels K_{ij} to application T_i.

Let $x_{ijk} = 1$ if task T_i is assigned quality point K_{ij} and resource consumption ρ_{ijk}, and $x_{ijk} = 0$ otherwise. Hence if task T_i is accepted for processing by the system, then exactly one of the indicator variables x_{ijk} equals 1; while the others are 0. If T_i is not accepted, then all are 0.

Using this notation, the optimization problem can be stated as shown in Fig. 4.

$$\textit{Maximize} \quad \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} x_{ijk} u_i(k_{ij})$$

$$\textit{subject to} \quad \sum_{i=1}^{n} \sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} (x_{ijk} \rho_{ijkl} \le r_l^{\max}), (l = 1,\ldots,m)$$

$$\sum_{j=1}^{|Q_i|} \sum_{k=1}^{N_{ij}} (x_{ijk} \le 1), (i = 1,\ldots n)$$

$$(x_{ijk} \in \{0,1\}), (i = 1,\ldots,n), (j = 1,\ldots,|Q_i|), (k = 1,\ldots, N_{ij})$$

Fig. 4. Formulation of the optimal solution to the MRMD problem.

Note the following.

- ρ_{ijkl} is the lth coordinate of the vector ρ_{ijk}.
- The possible QoS levels for application $T_i(K_{ij})$, their utility ($u_i(K_{ij})$), resource requirements using ρ_{ijk} (ρ_{ijkl}, $1 \le l \le m$) and total resource availability $(r_l^{\max})$ are given constants.
- The variables to be selected to optimize total system utility are the x_{ijk}.

The Q-RAM advisor provides a local search technique that is an approximate solution to MRMD. The Q-RAM solution is several orders of magnitude faster than the optimal dynamic programming and mixed integer programming solutions. The approximation algorithm allows the user to trade-off nearness to the optimal solution versus performance.[18]

8. Contracts and Probabilistic Guarantees

An Amaranth QoS contract is an agreement between the user and the system that the resources necessary to support the given QoS level will be provided for the duration of the session with a given probability. Ideally, the fixed QoS level would map to a fixed resource demand. But this is not the case for many real world applications such as video conferencing. Due to the effects of compression and dependent upon the amount of motion in the scene, the bandwidth requirements for a video stream may vary over time. Bandwidth requirements are highest when the camera is panning or zooming. They are lowest when the camera is focused on a low motion scene such as people sitting at a conference table. Network flows having these characteristics are often called Variable Bit Rate (VBR) flows.

With a system based on hard reservations, it is difficult to achieve high resource utilization due to the worst case assumptions that must be made for sessions with VBR

flows. Therefore, Amaranth provides these sessions with probabilistic guarantees that are specified in terms of a lower bound on the expected value(s) of one or more Probabilistic Level of Service (PLoS) metrics. Formulated as probabilities, PLoS values may vary continuously between 0 for sessions requiring only best-effort flows and 1 for sessions requiring hard reservation. The Amaranth system currently uses two PLoS metrics: (1) QoS Availability (fraction of time there is no degradation) and (2) the fraction of packets delivered (not dropped). Other PLoS metrics, such as the fraction of packets delivered on time, are feasible. A single session can specify requirements for more than one metric, and different sessions may use different metrics.

The RPM policy module and RPM kernel-level mechanisms implement the probabilistic guarantees in Amaranth. To enforce the probabilistic guarantees, the RPM kernel mechanisms maintain a set of priority modes. Each priority mode *j* is a strict ordering of the managed sessions. When there is an overflow of packets in a router's queue, the system drops packets of the low priority session associated with the active priority mode. By time multiplexing through the priority modes, the probabilistic level of service delivered to each of the sessions can be controlled by varying the mode holding times. The RPM policy module evaluates session requests to determine if sufficient resources are available and computes the set of mode holding time parameters needed by the kernel-level mechanisms to enforce the requested guarantee of service. One can represent the bandwidth resources required by a session as a Markov model with each state characterized by a probability and an instantaneous resource demand. One then formulates a linear program LP to determine the amount of time, α_j, that the system should be in each mode *j* in order to achieve the required PLoS value for each task *i*.

$$\min\left(\sum_{j=1}^{n}\alpha_j\right), such that\ \forall i, \left(\sum_{j=1}^{n}\rho_{ij}\alpha_j \geq A_i\right) \tag{1}$$

where *n* is the number of modes, *i* is the task number, ρ_{ij} is the estimated value of a particular PLoS metric for task *i* while in mode *j*, A_i is the required PLoS value for task *i*, and α_j is the fraction of time the system is in mode *j*.

Since we are only interested in satisfying the constraints and do not have another objective function, we modify LP to minimize the sum of the α_j rather than to specify it directly as a constraint. This modification, as shown in Eq. 1, helps us to avoid several numerical instability problems. The left-hand side of the inequality represents the PLoS delivered to task *i* in the interval corresponding to α_j. The right-hand side represents the PLoS requirement A_i for task *i*. All of the constraints are "greater than" constraints because the delivered PLoS must be at least as good as the required PloS. The coefficients of the constraints are the estimated PLoS values for a session while in a particular mode. The sum of the α_j must equal one. If the solution to the modified LP results in the sum of the α_j being less than or equal to one, then the original constraint problem is satisfiable. Since all of the inequality constraints have positive coefficients, we can safely increase the α_j values until they equal one without causing any of the constraints to be violated. If the solution to the modified LP results in the sum of the α_j being greater than one, then the original constraint problem is unsatisfiable; and we must reject the admission request or renegotiate for a lower PLoS or QoS setpoint.

Probabilistic guarantees can result in significant gains in resource utilization. For example, consider a 100 Mbps link used to carry a number of video conference flows. Suppose that each flow requires 1.5 Mbps 55% of the time, 2.7 Mbps 37% of the time, and 5 Mbps 8% of the time. Fig. 5 shows the number of flows that can be admitted on the link as a function of the QoS Unavailability (one minus the QoS Availability) for each of the flows. To provide hard guarantees on the flows, the system could admit no more than 20 flows (100 Mbps/5 Mbps). With probabilistic guarantees, the system can admit nearly double the number of flows because the system does not have to provide each flow with a 100% guarantee of service. In practice, probabilistic guarantees work whenever the likelihood of synchronized worst-case phasing of flows is small.

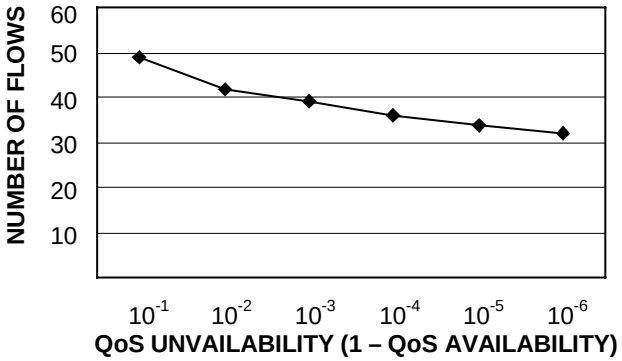

Fig. 5. An example of the feasible number of the flows with respect to QoS unavailability.

9. Reserve Capacity Approach to Admission Control and Resource Allocation

The Amaranth admission goal is to provide high utility consistent with honoring contracted assurance probabilities for QoS violation rates. To accomplish this goal, Amaranth researchers are experimenting with holding resources in reserve to handle random, but statistically likely, bursts of task requests. Using ideas borrowed from control theory, the system attempts to maintain reserve capacity at or near a predetermined value (the *setpoint*). The system compares the amount of actual unused capacity against the setpoint and, under the guidance of a control policy, determines the QoS level offered to new task admissions.

The admission control system works in two steps. First, the system admits the session if there are sufficient resources available for the application's minimum or baseline QoS requirement. For critical applications such as the battle group example discussed previously, the system is sized to be large enough to guarantee admission to all possible critical tasks at a minimum QoS level. Then the system uses the admission/resource allocation control parameters to determine the actual amount of resources to be allocated to the incoming task and allocates above the minimum level only in accordance with the admission policy.

Current experiments deal with five policies for network bandwidth allocations. Two simplistic policies are: (1) to always allocate the maximum bandwidth possible (a greedy "Best Effort" strategy) and (2) to always allocate the minimum requested bandwidth (a

pessimistic "Hard Reservation" strategy). Three common control theory policies complete the set of five policies: (3) Bang-bang control, (4) Proportional control, and (5) Proportional-Integral-Differential (PID) control. What is novel is not the control policies themselves but rather the approach of applying them to admission control for QoS management.

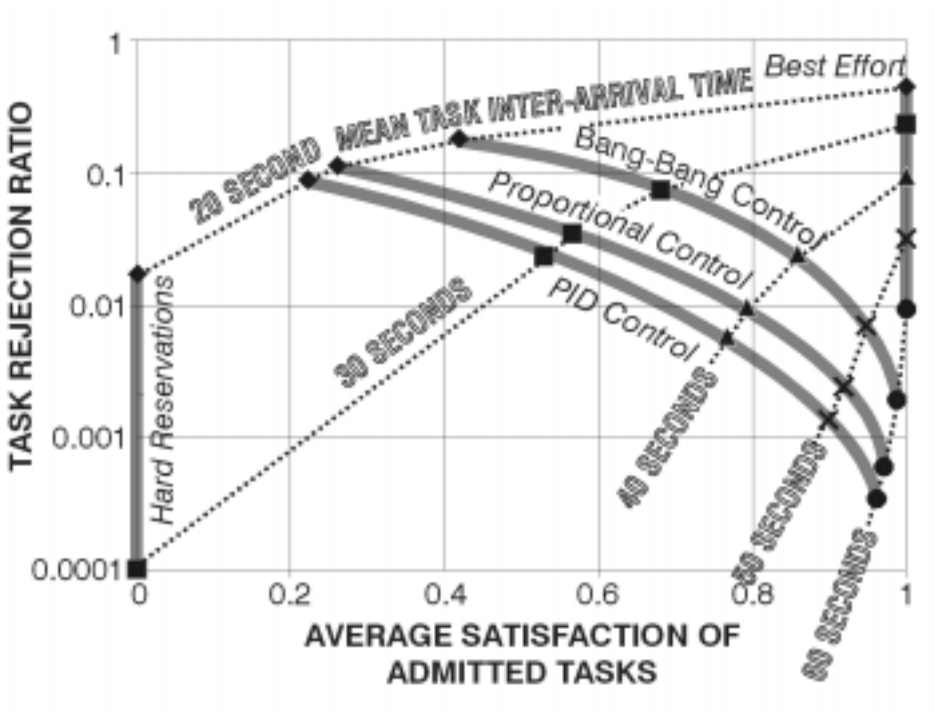

Fig. 6. Task rejection ration versus average satisfaction of admitted tasks for inter-arrival times (IA) 20 through 60 seconds and across various algorithms for allocating resources.

Fig. 6 shows simulation results for the five control policies applied across a range of task inter-arrival times. The task inter-arrival times (means shown in Fig. 6 by dashed lines) as well as the task durations of the simulated tasks (mean of 10 minutes for all experiments) are exponentially distributed. The resource demand is uniformly distributed with minimum and maximum bandwidth demands of [1,4) Mbps and (4,7] Mbps, respectively. The reserve capacity setpoint is 10 Mbps, and the total available bandwidth is 100 Mbps (i.e., the desired goal is to maintain 10% reserve capacity). A mean inter-arrival time of 20 seconds represents a heavily loaded resource, while 60 seconds represents a lightly loaded resource for this scenario. While these experimental parameters are merely exemplary values, they do provide an example of a system's response across a range of loading levels.

For experimental simplicity, the system rejects a task entirely if there is insufficient bandwidth to admit the task with its lowest acceptable QoS. (In a real system, alternate approaches are possible, such as degrading tasks with higher-than-minimum QoS level to make room for critical task admissions.) Fig. 6 shows the task rejection ratio versus the

average satisfaction of admitted tasks. The average task satisfaction is computed as shown in Eq. 2, where *alloc* is the allocated bandwidth, *min* is the minimum acceptable bandwidth, and *max* is the maximum requested bandwidth.

$$AverSat = \frac{(alloc - \min)}{(\max - \min)} \tag{2}$$

There are five solid curves in Fig. 6 corresponding to the performance of various control policies. For any given contracted assurance level, one can use the graph to indicate viable control policies and their expected average satisfactions (utility values under these simplified assumptions). This is done by considering portions of dashed lines below the task rejection ratio having the value of (1-contracted assurance level). For example, if a particular system must achieve an assurance level of 99% with a mean arrival rate of 40 seconds, it can do so by using proportional control, PID control, or hard reservations. Proportional control provides the best average satisfaction in this scenario (bang-bang control provides higher satisfaction but has a task rejection ratio higher than 0.01).

In general, the trade-off seen in these results is the expected one of lower rejection levels coming at the price of lower satisfaction levels. However, the use of admission policies based on control theory provides intermediate trade-off points beyond those available with either hard reservations or best-effort policies. In comparison to the simplistic bang-bang control policy, the more sophisticated control policies appear to better anticipate and avoid downstream problems caused by overly aggressive QoS allocation, while achieving lower rejection ratios for a given task arrival rate, than overly conservative schemes.

10. Results and Conclusions

Prototypes for most of the components are complete as well as the test of most component interactions. The Amaranth simulator currently simulates the RPM scheduler as well as user sessions and Amaranth-controlled network allocations and routing. D-MOD is able to detect and predict node failures, and FRUM is able to track and record resource usage patterns on nodes and their connected network links. The local Sesco is functional as well as the Q-RAM adviser module. The theoretical models for Q-RAM, RPM, and RTQM have been developed and validated manually or via simulation.

The goal for high-assurance computing is to guarantee that target application task arrivals receive at least minimum or baseline QoS. The system designer can use the application and resource models to size the system to have sufficient resources to satisfy the worst case arrival of all tasks with each critical and non-critical task receiving at least its minimum required resources. Alternatively, the system designer can size the system to handle the worst case arrival of critical tasks. The critical tasks would receive at least minimum QoS, while the non-critical tasks whose total weighted utility maximizes the overall system utility without exceeding the available resources would receive resource allocations. In such high-assurance systems, QoS violations would comprise degrading previously admitted tasks to minimum QoS rather than denying admission to critical tasks.

11. Summary and Future Research

This paper has discussed the Amaranth framework for providing policy-based QoS management with probabilistic guarantees that a QoS contract will be upheld. The naval battle group application, presented in this paper, requires a high degree of assurance that necessary system resources will be allocated to critical mission activities while maximizing the utilization of available system resources. Via diagrams and text, the paper has described the functionality and structure of the Amaranth QoS management system and architecture. Lastly, the paper has overviewed the Q language for specifying QoS parameters, utility-based QoS management, contracts and probabilistic guarantees, and admission control using a reserve capacity approach.

In addition to the utility-based QoS management policy, the Amaranth system will support other policies for managing resources from the user as well as the system viewpoints. For example, current research has yielded an innovative way to integrate the utility and reserve capacity based approaches to resource allocation. Implementation of the RPM method for providing probabilistic guarantees and work on the integration of real-time queueing theory with the determination of optimal utility across current application requests are ongoing. The design of the Global Sesco Coordinator to scale to interconnected LANs and WANs with local sets of nodes and links each managed by a different local Sesco is part of future project research. Most importantly, the Amaranth framework enables the system manager to employ innovative resource management policies while maintaining probabilistic QoS guarantees and dependable communications.

Acknowledgments

The authors would like to acknowledge the contributions of the other members of the Amaranth Project as follows: Arjun Cholkar, LeMonte' Green, Pradeep K. Khosla, Chen Lee, John Lehoczky, Don Madden, Ragunathan Rajkumar, Lui Sha, Ying Shi, Daniel P. Siewiorek, and James Washburn.

References

1. R. Bittel, et al., "Soldier Phone: An Innovative Approach to Wireless Multimedia Communications," in *Proc. IEEE Symposium on Application-Specific Systems and Software Engineering & Technology* (IEEE Computer Society, Los Alamitos, CA, 1999), pp. 264-268.
2. A. Campbell, G. Coulson, and D. Hutchison, "A Quality of Service Architecture," in *ACM SIGCOMM Computer Communication Review* **2(24)**, 6-27 (Apr. 1994).
3. C. Aurrecoechea, A. Campbell, and L. Hauw, "A Survey of QoS Architectures," in *ACM/Springer Verlag Multimedia Systems Journal*, Special Issue on QoS Architecture **3(6)**, 138-151 (May 1998).
4. K. Nahrstedt and J. M. Smith, "The QoS Broker," in *IEEE Multimedia Magazine* **1(2)**, 53-61 (1995).
5. Z. Deng and J. W. S. Liu, "Scheduling Real-Time Applications in an Open Environment," in *Proc. IEEE 18th Real-Time Systems Symposium* (IEEE Computer Society, Los Alamitos, CA, Dec. 1997), pp. 308-319.
6. D. Hull, et al., "An End-to-End QoS Model and Management Architecture," in *Proc. IEEE Workshop on Middleware for Distributed Real-time Systems and Services* (IEEE Computer Society Press, Los Alamitos, CA, Dec. 1997), pp. 82-89.
7. T. F. Abdelzaher and K. G. Shin, "QoS Provisioning with qContracts in Web and Multimedia Servers," in *Proc. 20th IEEE Real-Time Systems Symposium* (IEEE Computer Society Press, Los Alamitos, CA, Dec. 1999), pp. 44-53.

8. J. Sydir, S. Chatterjee, and B. Sabata, "Providing End-to-End QoS Assurances in CORBA-Based Systems," in *Proc. First IEEE Inter. Symposium on Object-Oriented Real-Time Distributed Computing* (IEEE Computer Society Press, Los Alamitos, CA, Apr. 20-22, 1998), pp. 53-61.
9. B. Sabata, et al., "Taxonomy for QoS Specifications," in *Proc. Third Workshop on Object-Oriented Real-Time Dependable Systems* (IEEE Computer Society, Los Alamitos, CA, Dec. 1997), pp. 100-107.
10. S. Chatterjee, et al., "Modeling Applications for Adaptive QoS-based Resource Management," in *Proc. Second IEEE High-Assurance Systems Engineering Workshop* (IEEE Computer Society, Los Alamitos, CA, 1997), pp. 194-201.
11. M. B. Davis and J. J. Sydir, "Resource Management for Complex Distributed Systems," in *Proc. Second Workshop on Object-Oriented Real-Time Dependable Systems (*IEEE Computer Society, Los Alamitos, CA, 1996), pp. 113-115.
12. A. Bashandy, et al., "A Protocol Architecture for Guaranteed Quality of Service in Collaborated Multimedia Applications," in *Proc. IEEE Symposium on Application-Specific Systems and Software Engineering & Technology* (IEEE Computer Society, Los Alamitos, CA, 1999), pp. 120-127.
13. I.R. Chen and T.H. Hsi, "Performance Analysis of Admission Control Algorithms Based on Reward Optimization for Real-Time Multimedia Servers," in *Performance Evaluation* **2(33)**, 89-112 (Jul. 1998).
14. A. DeWitt, et al., "ReMoS: A Resource Monitoring System for Network-Aware Applications," CMU-CS-97-194, Dec. 1997.
15. C. Lee, J. Lehoczky, R. Rajkumar, and D. Siewiorek, "On Quality of Service Optimization with Discrete QoS Options," in *Proc. Fifth IEEE Real-Time Technology and Applications Symposium* (IEEE Computer Society, Los Alamitos, CA, 1999), pp. 276-286.
16. R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "Practical Solutions for QoS-based Resource Allocation Problems," in *Proc. 19th IEEE Real-Time Systems Symposium* (IEEE Computer Society Press, Los Alamitos, CA, Dec. 1998), pp. 296-306.
17. R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A Resource Allocation Model for QoS Management," in *Proc.18th IEEE Real-Time Systems Symposium* (IEEE Computer Society Press, Los Alamitos, CA, Dec. 1997), pp. 298-307.
18. C. Lee, J. Lehoczky, D. Siewiorek, R. Rajkumar, and J. Hansen, "A Scalable Solution to the Multi-Resource QoS Problem," CMU/CS 99-144, May 1999.
19. J. P. Lehoczky, "Scheduling Communication Networks Carrying Real-Time Traffic," in *Proc. 19th IEEE Symposium on Real-Time Systems* (IEEE Computer Society, Los Alamitos, CA, 1998), pp. 470-479.
20. J. P. Lehoczky, "Real-Time Queueing Network Theory," in *Proc. 18th IEEE Real-Time Systems Symposium* (IEEE Computer Society Press, Los Alamitos, CA, 1997), pp. 58-67.
21. J. P. Lehoczky, "Using Real-time Queueing Theory to Control Lateness in Real-Time Systems," in *Performance Evaluation Review* (Jun. 25, 1997), pp. 158-168.
22. J. P. Lehoczky, J.P., "Real-Time Queueing Theory," in *Proc. 17th IEEE Real-Time Systems Symposium* (IEEE Computer Society Press, Los Alamitos, CA, 1996), pp. 186-195.
23. P. Chandra, et al., "Darwin: Resource Management for Value-Added Customizable Network Service," in *Proc. Sixth IEEE International Conference on Network Protocols* (IEEE Computer Society Press, Los Alamitos, CA, Oct. 1998), pp. 177-188.
24. Tcl (Tool Command Language) is a scripting language that can be used either as a stand-alone software application or as software commands embedded in application programs. Tk is a graphical user interface toolkit for the rapid creation of graphical user interfaces. Tcl and Tk are available from http://www.tcltk.com/, accessed 8 Oct. 2000.
25. J. P. Loyall, et al., "Specifying and Measuring Quality of Service in Distributed Object Systems," in *Proc. First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (IEEE Computer Society Press, Los Alamitos, CA, Apr. 20-22, 1998).

26. Vic is a video conferencing software application developed by the Network Research Group at the Lawrence Berkeley National Laboratory in collaboration with the University of California, Berkeley. For more information, see http://ftp.ee.lbl.gov/vic/, accessed 8 Oct. 2000.
27. H.261 is a standard for digitizing and compressing analog video. Users typically apply it to video conferencing applications. The International Telecommunications Union (ITU) developed the standard. See the ITU website at http://www.itu.int, accessed 8 Oct. 2000, for a copy of the H.261 recommendation.

Carol L. Hoover received the BS and MS degrees in Computer Science from the University of Akron and Ohio State University in 1985 and 1987, respectively, after teaching in the public schools. She was a computer manager in a US Congressional office and a graduate research intern at NASA Lewis Space Center. From 1988-1992, she developed factory automation software for the Allen-Bradley Company and was a senior software engineer. At Carnegie Mellon University since 1992, she was a lecturer in real-time systems for the Masters of Software Engineering Program in the School of Computer Science and a project scientist in the Robotics Institute. She currently researches the design of high-assurance software systems as a doctoral candidate in Electrical and Computer Engineering. She is a member of the ACM and IEEE.

Jeffery Hansen received the BS degree in Electrical Engineering, Computer Engineering and Mathematics from Carnegie Mellon University in 1987, and the MS and PhD degrees in Electrical and Computer Engineering also from Carnegie Mellon University in 1988 and 1992, respectively. He has been a research scientist with the Institute for Complex Engineered Systems at Carnegie Mellon University since 1998 where he is doing research in quality of service for distributed systems. Before 1998, he was a research scientist at the Toshiba ULSI Engineering Laboratory in Kawasaki, Japan.

Philip Koopman received the BS and MEng degrees in computer engineering from Rensselaer Polytechnic Institute in 1982. After serving as a US Navy submarine officer, he returned to graduate school and received the PhD degree from Carnegie Mellon University in 1989. During several years in industry, he was a CPU designer for Harris Semiconductor and an embedded systems researcher at United Technologies Research Center. He has been at Carnegie Mellon University since 1996, where he is currently an assistant professor. His research interests include distributed embedded systems, dependability, and system architecture. He is an associate editor of *Design Automation for Embedded Systems: An International Journal*. He is also a member of the ACM and a senior member of the IEEE.

Sandeep Tamboli is a software development engineer at Marconi Communications, USA. He completed his MS in the Electrical and Computer Engineering Department at Carnegie Mellon University in August 2000. He was a graduate research assistant working on the Amaranth project under the supervision of Dr. Philip Koopman. His Master's research involved the evaluation of different admission policies for probabilistic QoS at the application level. From 1996-1998, he was a systems programmer analyst with Information Technology Group at Texas Instruments.