Towards Integrated Runtime Solutions in QoS-aware Middleware

Baochun Li
Department of Electrical and Computer Engineering
University of Toronto
bli@eecg.toronto.edu

Dongyan Xu, Klara Nahrstedt
Department of Computer Science
University of Illinois at Urbana-Champaign
d-xu,klara@cs.uiuc.edu

ABSTRACT

Future-generation multimedia applications are expected to be highly *scalable* to a wide variety of heterogeneous devices, and highly *available* across wide-area distributed environments. This demands multiple stages of run-time support in QoS-aware middleware architectures, particularly, *probing* the performance of QoS parameters, *instantiating* the initial component configurations, and *adapting* to on-the-fly variations. In this paper, we review past experiences and lessons learned in our existing architectures with respect to run-time support, and present a novel approach to unify these stages into an integrated run-time middleware architecture, so that multimedia applications are monitored and configured in a coherent fashion.

1. INTRODUCTION

With the advent of next-generation multimedia technologies such as very-low bit rate MPEG-4 streaming and voice-over-IP, future multimedia applications are expected to be highly *scalable* to a wide variety of heterogeneous devices, and highly *available* across wide-area distributed environments. On the other hand, current-generation distributed multimedia applications (such as video-on-demand, multimedia streaming and visual tracking) are developed in an ad-hoc fashion, while their performances are tailored to specific operating systems and platforms.

It has been envisioned in recent work [1] that with middleware architectures designed for Quality-of-Service (QoS) requirements in multimedia applications, the following unique characteristics in pervasive and heterogeneous environments are feasible for these applications: First, once componentized to *multimedia services* and *consumers*, various *service configurations* are possible, and may be selected by the middleware based on resource availability and user preferences at the time of application instantiation; Second, multimedia applications are subject to both off-line[1] and run-time probing with respect to the performance of their QoS parameters, and such QoS probing is the responsibility of middleware components; Third, during application run-time, the middleware layer may attempt to assist the application to adapt to 'triggering sources' including changing user requirements or resource availability. Such 'triggering sources' are often caused by statistical multiplexing of concurrent resource usage; lack of resource reservation schemes; or user preference/environment changes such as user mobility. To summarize, the QoS-aware middleware is most effective in supporting multimedia applications during application run-time, by *probing*, *instantiating* and *adapting* applications, tailoring their performances to user behavior and pervasive environments.

In our recent work, we have focused on several different aspects of QoS-aware middleware that was previously summarized. The *Agilos* middleware project [3] has focused on the aspects of off-line probing and run-time adaptation, particularly how to make informed decisions on when, how and to what extent to adapt to the fluctuations in resources and user requirements. In comparison, the $2K^Q$ project [4] has focused on architectures and protocols to understand user requirements and appropriately instantiate a particular service configuration for the application, so that minimum user-specified requirements are met. In this paper, we evaluate our past experiences and learned lessons with previously proposed middleware architectures, and make the following three key observations. First, there exist three critical aspects of **run-time support** that the middleware components are in the best positions to provide: *probing*, *instantiating* and *adapting* multimedia applications. Second, it has not previously been identified that the 'driving force', or 'triggering sources', of these three critical aspects are *identical*. They include variations in user preferences and resource availability. Third, by integrating run-time solutions with respect to *probing*, *instantiating* and *adapting* to the unified 'triggering sources', we may design a unified decision-making process to configure and adapt the applications in a coherent fashion.

There exist a number of proposed frameworks and systems for the purpose of managing run-time resource usage and application component configuration at the middleware level. For example, the BBN *QuO* project [5] has proposed an adaptation-based middleware-level architectural enhancement to CORBA; the *Da CaPo++* [6] framework has implemented adaptation-based services in the middleware; in the *Adapt* project [7], the concept of *open binding* was introduced as a programming model for the implementation of adaptation policies in mobile multimedia applications; in the *Darwin* project [8], a hierarchical service brokerage architecture was proposed for composing complicated and value-added distributed services.

This paper attempts to identify such common 'triggering sources' for different stages of run-time solutions, and propose the design of a coherent architecture to unify the run-time phases of probing,

[1] Probing, or *monitoring*, applications in benchmarking runs under 'sand-boxed' environments to emulate resource availability. Refer to [2] for details.

ACM Multimedia Middleware Workshop '01, Ottawa, Canada

instantiating and adapting applications. The rest of the paper is organized as follows. Section 2 summarizes and evaluates our past experiences with respect to run-time support in QoS-aware middleware. Section 3 identifies common 'triggering sources' for activating these run-time solutions. Section 4 proposes a coherent architecture to integrate different stages of run-time solutions. Section 5 concludes the paper with directions about future work.

2. PAST EXPERIENCES WITH QOS-AWARE MIDDLEWARE DESIGN

In the past several years, we have concurrently developed two separate middleware designs for supporting application-level Quality-of-Service, namely, the $2K^Q$ and *Agilos* projects, with different design objectives and assumptions. The $2K^Q$ project [4] aims at QoS provisioning with the presence of OS-level resource reservation mechanisms, particularly focusing on two important aspects. First, before run-time initialization, it translates between application-level and OS-level QoS parameters; Second, during application run-time, it instantiates a specific service configuration by checking both resource availability (via a service configuration selection algorithm) and available services at run-time (via a resource-aware service discovery mechanism). In comparison with $2K^Q$, the *Agilos* project [3] does not assume the presence of existing resource reservation schemes at the OS level, and emphasizes QoS adaptation mechanisms at run-time, caused mainly by resource variations. It resorts to off-line probing techniques to obtain the relationships between application-level QoS parameters and their corresponding resource demands, and uses a rule-based system to cater to application-specific needs for specifying adaptation preferences. In addition, the internal processing engine for generating adaptation decisions is generic and application-neutral, designed using control-theoretical techniques. Figure 1 shows the skeleton architecture for both projects, highlighting their differences in design objectives and assumptions with respect to run-time operations.

As illustrated in Figure 1, in both projects we use a generic *application component model*, shown as an *application component graph*. In this model, we view a collection of interconnected *application components* on a single host as a set of tasks, with input-output dependencies. Beyond a single end host, we group the entire distributed application into *clients* and *services*. The collection of clients and services form another directed graph representing the service provider-consumer relations.

Based on this model, middleware components of both projects attempt to *instantiate* or *reconfigure* the application component graph based on certain "triggering sources". In the $2K^Q$ case, the descriptors of "service requests" at application initialization time are translated to resource-level QoS requirement vectors, which are used to select a specific application component graph (i.e., a *service configuration*), and subsequently reserve resources on each of the hosts in the component graph. As a comparison, in the *Agilos* case, the resource-level control values are processed against an application-specific rule base, generating application-specific control actions as outputs. Such generated control actions may include tuning specific parameters within a component, or reconfiguring the application component graph.

With respect to *QoS probing*, the $2K^Q$ project includes *resource monitors* to perform end-to-end resource checking in order to select an appropriate service configuration. Such resource checking are performed at application instantiation time. The *Agilos* project, on the other hand, actively monitors resource usage via the *Observers* during application run-time, and makes adaptation decisions based on such run-time probing. In addition, a separate component, the *QualProbe*, is used to perform off-line probing during benchmarking runs, in order to discover the mapping between application-level and resource-level QoS parameters, thus assisting the specification of the rule base.

Observing from the previous discussions, it is noted that the "driving forces" and the "processing engines" of both middleware designs are strikingly similar with respect to their run-time support. It is therefore preferable to integrate these run-time solutions and present a unified architecture, in order to simply the design and accelerate the acceptance. We then proceed to discuss our proposals towards this goal.

3. TRIGGERING SOURCES TO ACTIVATING RUN-TIME SUPPORT

Before we present our design of an integrated run-time support architecture, we need to first identify the triggering sources, or "driving forces", that activates such run-time support. Towards this end, rather than attempting a "cure-all" middleware solution for any applications, we consider a typical range of distributed multimedia applications that may easily be componentized and modeled with the application component graph. These applications include video multicast streaming services, video-over-IP telephony, video conferencing or visual tracking applications. They may be deployed over a variety of computing devices, from PDAs to clustered workstations. We have identified that activating run-time middleware support to react to the following types of changes are necessary:

1. **Variations in resource availability.** Such variations are generally shown as changes in specific resource-level QoS parameters. We are most concerned with the following parameters. (1) client-side parameters such as CPU availability, bandwidth at the inbound interface for multimedia streaming, or buffer space. (2) service-side parameters of similar types; (3) parameters that show network performances between the service and the client, such as delay, jitter and loss. In the application component graph, all three categories of parameters may be represented as a "label" on either the component itself (core parameters), the inbound and outbound interfaces of the component (inbound and outbound parameters), or on an edge of the component graph (edge parameters). If these labels are shown as <parameter type, parameter value> pairs, Figure 2 illustrate an example. These <type, value> pairs are coherently monitored by on-line QoS probing components that execute in each of the hosts, usually by interacting with the OS kernel via system calls.

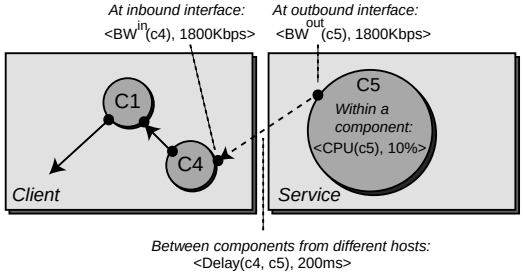

Figure 2: Labeling the Application Component Graph

2. **Variations in user preferences.** There are two categories of preferences that a user may specify and change at run-time. First, the level of satisfaction (e.g. Quality of Perception in

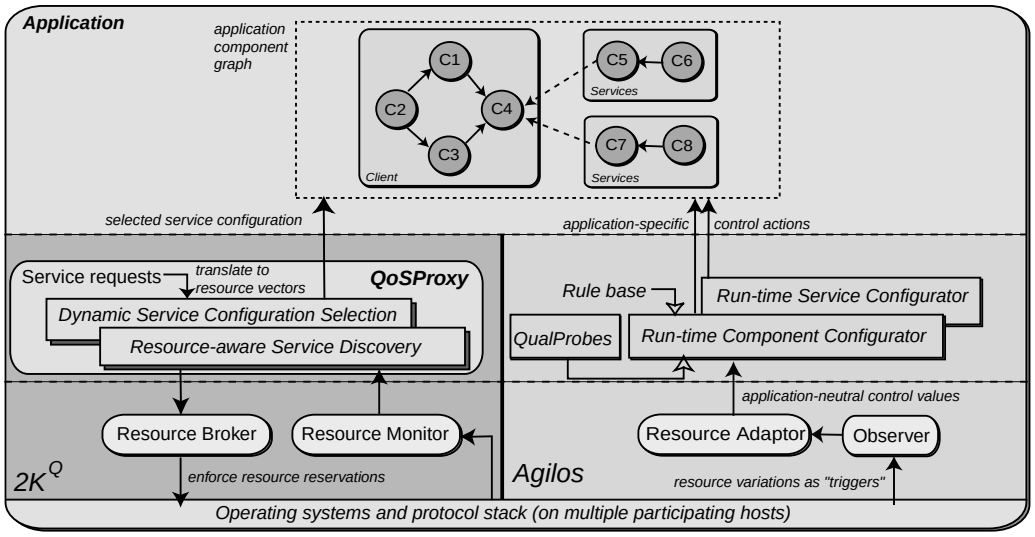

Figure 1: A run-time comparison between the $2K^Q$ and *Agilos* middleware architectures

video streaming clients) at different application QoS levels; second, the tradeoff policy among QoS parameters of conflicting interests. For example, for video streaming, one user may require the best image quality with lower frame rate, while another user may prefer the highest frame rate with lower image quality.

3. **User mobility.** In a complex distributed environment, the mobility of users should be treated as a normal case, rather than as an exception. Since the user may move with the same client host or with multiple client hosts, and then reconnects at the new location (either wirelessly or with cables), the application component graph may be disrupted (changed) for a short period of time, referred to as *application-level handoff*. If we consider the "location"[2] of an application component to be one of its core parameters, It is observed that any user mobility may be treated as a either a *spontaneous reconfiguration* in the application component graph or parameters that are already labeled in such a graph.

It is apparent that if we only consider the above three types of triggering sources that may activate run-time support, we may conveniently use the application component graph and its parameters (core, edge, inbound or outbound) to effectively model such triggering sources. In other words, we may define a data structure, referred to as *application states*, maintained in all participating hosts, which includes the following information: (I) the definition of the *application component graph*; (II) possible instances of alternative service configurations that may result from topological changes in the application component graph. These are represented by different topologies of the same pool of components; (III) related parameters that are labeled in such a graph; (IV) upper and lower bounds of these relevant parameters. Any changes in such a data structure (particularly parts I and III) will activate run-time middleware behavior; while any run-time behavior may also influence these application states. Such application states are the basis of the discussions on our integrated architecture as follows.

[2]The *location* of a component may be represented by its host's address or interface name.

4. AN INTEGRATED RUN-TIME SUPPORT ARCHITECTURE

We propose an integrated middleware architecture for run-time application support, specializing in run-time probing (monitoring) of application states, run-time instantiation of a specific service configuration, and run-time adaptation to application state variations. For all three types of support, we observe identical "triggering sources", namely, *application states* maintained on all participating end hosts in the distributed application. The main design can be summarized as the following.

(a) For the purpose of *QoS probing and monitoring* of application states, each participating host activates a middleware component, referred to as the *"probe"*, at fixed intervals. When activated, it goes through three probing steps. First, it checks input-output relationships of current application components against cached application states (which include an *application component graph*). If there exists a mismatch, it first updates its own cached application states to reflect the discovered changes, then multicasts such changes to all other participating hosts. Second, it checks the input-output relationship between all local application components and relevant components of other hosts. It updates its cached states correspondingly if there is a change. The first two steps are responsible of detecting topological changes in the application component graph. The final step is to use OS-level system calls or application-level hooks to measure all labeled parameters (including inbound, outbound, core and edge parameters) that are related to the residing host of the "probe". The probing results are stored back in the application states, and multicasted to all participating hosts if necessary.

(b) For the purpose of *run-time instantiations*, we reuse the original design of the 'processing engine' in the $2K^Q$ architecture, but based solely on the *application states* stored on each participating host as the input. As in $2K^Q$, such run-time instantiation process is divided into two parts. It first executes a particular *service discovery* algorithm to discover unknown services at application compile-time, followed by a *dynamic service configuration selection algorithm* to finalize the actual service configuration to be enforced by the middleware.

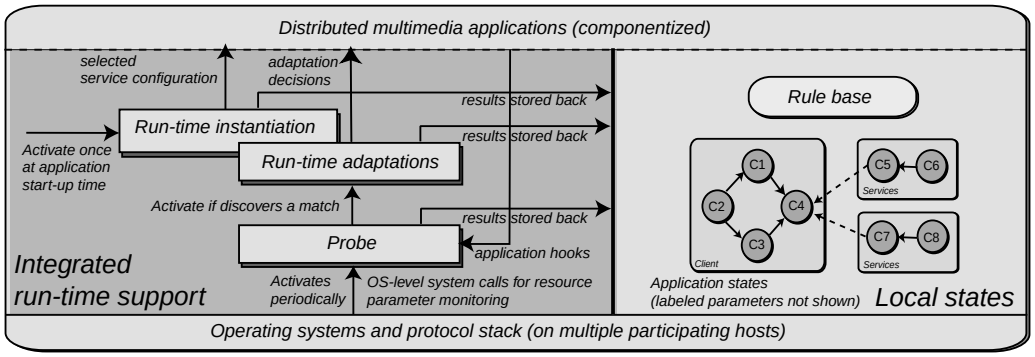

Figure 3: An Integrated Run-time Support Architecture

Such enforcement effectively changes the application states, the results of which are stored back into participating hosts after the run-time instantiation stage.

(c) For the purpose of *run-time adaptations*, again, we reuse our original design of the *configurators* from the *Agilos* architecture. In order to represent the policies for such adaptations, we utilizes a particular application-specific rule-based system based on application states. This rule-based system is critical to the entire process of automatically performing run-time adaptation tailored to the needs of a specific application. For example, assume the "probe" on a specific host is activated on adjacent time instants t_1 and t_2, while the application states measured on such instants are $S(t_1)$ and $S(t_2)$, respectively. A specific rule in the rule-based system is represented as a standard "if ... then ..." clause, with the precondition being $p_{S(t_1)-S(t_2)}$, where p is either a labeled parameter or the topology of the component graph. In the case of a parameter, it represents the scalar differences of such a parameter in between $S(t_1)$ and $S(t_2)$; otherwise, it shows the topological change between time t_1 and t_2 in the component graph. The results of such adaptations are represented by a modified application state, illustrated by either parameter changes or topological changes in the graph. Again, such results are stored back into participating hosts after performing the adaptation. Such adaptation behavior is activated by the "probe" only when necessary, when it discovers a match between its observed application state and the precondition of one of the rules in the rule base.

We summarize the above discussions of such an integrated run-time middleware architecture in Figure 3. As illustrated, the tenet of such a design is to propose a simple, yet effective, architecture operating on clearly identified triggering sources.

5. CONCLUDING REMARKS

In this paper, we have presented a unified middleware architecture to integrate various run-time solutions found in previous middleware designs with different perspectives, namely, $2K^Q$ and *Agilos* projects. Such integration leads to a streamlined design, so that applications may be configured at run-time in a coherent fashion. We have identified three common sources of triggering such run-time middleware support, followed by proposing the design of the unified architecture. With our years of experiences designing and implementing QoS-aware middleware components, we have come to believe that integrating and streamlining a wide variety of existing solutions into a coherently designed, simple framework is critical to the acceptance of such middleware in mainstream computing. Such framework does not have to be a "cure-all" solution; rather, it needs to identify an array of distributed multimedia applications of similar categories, and is only customized towards the needs of these applications. Implementation of such a streamlined, integrated run-time framework is still on-going work, with multimedia multicast streaming as the application of focus. The objective of this paper is to discuss the merits of such a design, learning from our past experiences with QoS-aware middleware.

6. REFERENCES

[1] X. Fu, W. Shi, A. Akkerman, and V. Karamcheti, "CANS: Composable, Adaptive Network Services Infrastructure," *USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2001.

[2] F. Chang and V. Karamcheti, "A Framework for Automatic Adaptation of Tunable Distributed Applications," *Cluster Computing: The Journal of Networks, Software and Applications*, 2001.

[3] B. Li and K. Nahrstedt, "A Control-based Middleware Framework for Quality of Service Adaptations," *IEEE Journal of Selected Areas in Communications, Special Issue on Service Enabling Platforms*, vol. 17, no. 9, pp. 1632–1650, September 1999.

[4] K. Nahrstedt, D. Wichadakul, and D. Xu, "Distributed QoS Compilation and Runtime Instantiation," *In Proceedings of the Eighth IEEE/IFIP International Workshop on Quality of Service*, pp. 198–207, June 2000.

[5] J. Zinky, D. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, 1997.

[6] B. Stiller, C. Class, M. Waldvogel, G. Caronni, and D. Bauer, "A Flexible Middleware for Multimedia Communication: Design, Implementation, and Experience," *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 9, pp. 1580–1598, September 1999.

[7] T. Fitzpatrick, G. Blair, G. Coulson, N. Davies, and P. Robin, "Supporting Adaptive Multimedia Applications through Open Bindings," in *Proceedings of International Conference on Configurable Distributed Systems (ICCDS '98)*, May 1998.

[8] P. Chandra, A. Fisher, C. Kosak, T. Ng, P. Steenkiste, E. Takahashi, and H. Zhang, "Darwin: Resource Management for Value-Added Customizable Network Service," in *Proceedings of IEEE International Conference on Network Protocols (ICNP '98)*, October 1998.