Multimedia Systems 8: 420–430 (2002)
Digital Object Identifier (DOI) 10.1007/s00530-002-0061-4

Multimedia Systems

An integrated runtime QoS-aware middleware framework for distributed multimedia applications

Baochun Li[1], **Dongyan Xu**[2], **Klara Nahrstedt**[3]

[1] Department of Electrical and Computer Engineering, University of Toronto, Canada (e-mail: bli@eecg.toronto.edu)
[2] Department of Computer Sciences, Purdue University, USA (e-mail: dxu@cs.purdue.edu)
[3] Department of Computer Science, University of Illinois at Urbana-Champaign, USA (e-mail: klara@cs.uiuc.edu)

Abstract. Future-generation distributed multimedia applications are expected to be highly scalable to a wide variety of heterogeneous devices, and highly adaptive 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. However, few of the past experiences in related work have shown comprehensive run-time support in all of the above stages – they often design and build a middleware framework by focusing on only one of the run-time issues. In this paper, we argue that distributed multimedia applications need effective run-time middleware support in all these stages to be highly scalable and adaptive across a wide variety of execution environments. Nevertheless, the design of such a middleware framework should be kept as streamlined and simple as possible, leading to a novel and integrated run-time middleware platform to unify the probing, instantiation and adaptation stages. In addition, for each stage, the framework should enable the interaction of peer middleware components across host boundaries, so that the corresponding middleware function can be performed in a coordinated and coherent fashion. We present the design of such an integrated architecture, with a case study to illustrate how it is simple yet effective to monitor and configure complex multimedia applications.

Key words: Run-time adaptation, middleware support, multimedia systems

1. Introduction

With the advent of next-generation multimedia technologies such as mobile and ubiquitous media delivery, future multimedia applications are expected to be highly scalable to a wide variety of heterogeneous devices, and highly adaptive across wide-area distributed environments. Such objectives call for significant extensions to the current-generation distributed multimedia technologies, such as the streaming, caching, and processing of media data. However, these technologies are usually developed independently, and their performances are tailored and tuned to specific operating systems and platforms. On the other hand, it has been envisioned in recent work [1] that, with the assistance of multimedia middleware, the goals of achieving scalability and adaptivity can be achieved with minimum modifications to current-generation multimedia applications.

By interacting with operating system kernels and application-level hooks, QoS-aware middleware has been proved highly effective in supporting multimedia applications via run-time probing, instantiation and adaptation of applications. Therefore, application configurations and performances can be tailored to different user behavior and characteristics of ubiquitous environments. To be more specific, the QoS-aware middleware framework is expected to provide the following critical functions:

- *Run-time probing.* Distributed multimedia applications are subject to both off-line and run-time probing with respect to the instantaneous performances of their QoS parameters. Such QoS probing is the responsibility of middleware components. Probing is useful in learning about application behavior, so that better run-time adaptation rules may be set accordingly.
- *Run-time instantiation.* A distributed multimedia application may have various application configurations. Each application configuration consists of a set of application components; and is represented by a configuration graph. At the stage of run-time instantiation, the middleware will select a suitable configuration which matches the specific resource availability and user preference.
- *Run-time adaptation.* During application run-time, the middleware may assist the application to adapt to the *triggering sources*, including changes of resource availability or user requirements. Such triggering sources exist due to the sharing of resources among applications; lack of resource reservation mechanisms; or change of user preferences (including location and environment). The middleware changes the application behavior correspondingly when significant variations in these triggering sources are detected.

Past experiences (Sect. 2) have focused on various aspects of providing run-time middleware support. As two examples, the *Agilos* middleware project [2] has focused on the aspects of off-line probing and run-time adaptation, particularly on how to make informed decisions regarding when, how and to what extent to adapt to the fluctuations in resources and user requirements. In comparison, the $2K^Q$ project [3] has focused on architectures and protocols that appropriately instantiate a particular application configuration for the application, so that minimum user-specified requirements are met.

On evaluating past experiences in related work, we make the following three key observations. First, few of the existing projects has shown integrated run-time support in all of the above aspects – they are typically designed with considerations of only one of the run-time issues. We argue that, to be highly scalable and adaptive across a wide variety of execution environments, distributed multimedia applications need effective run-time middleware support in all of these aspects. That said, the design of such middleware framework should be kept as streamlined and simple as possible, leading to a novel and integrated run-time middleware platform to unify the probing, instantiation and adaptation stages. Secondly, the design of such framework should consider the interaction of peer middleware components across host boundaries, to support distributed application components in a coordinated fashion. Third, it has not been previously identified that the triggering sources of these three critical functions are in fact identical. The triggering sources include variations in user preferences and resource availability. Therefore, run-time probing, instantiating and adapting can all be considered as *actions* driven by the same set of triggering sources. This leads to a unified and streamlined decision-making process to configure and adapt distributed applications in an integrated fashion. In this paper, we present the design of such an integrated architecture, with a case study to illustrate how it is simple yet effective to monitor and configure complex multimedia applications. Towards this objective, this paper identifies common run-time triggering sources for different run-time middleware functions.

The rest of the paper is organized as follows. Section 2 summarizes and evaluates past experiences in related work and our recent projects, with respect to run-time support by QoS-aware middleware. Section 3 identifies common triggering sources for activating run-time middleware functions. Based on these triggering sources, Sect. 4 proposes a unified architecture to integrate the middleware functions to be performed in different run-time stages. Section 5 presents an example multimedia application (video streaming and object tracking) as a case study, to show the effectiveness and simplicity of using the proposed framework to monitor and configure complex applications at run-time. Finally, Sect. 6 concludes the paper with directions to future work.

2. Past experiences in related work

There are 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 [4] proposed an adaptation-based middleware-level architectural enhancement to CORBA; the *Da CaPo++* [5] framework implemented adaptation-based services in the middleware; in the *Adapt* project [6], 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 [7], a hierarchical service brokerage architecture was proposed for composing complicated and value-added distributed services. From an architecture point of view, the group of work on reflective middleware designs [8] attempts to increase the flexibility of existing middleware frameworks by allowing them to be dynamically configurable to accommodate rapidly changing execution requirements.

In the past few years, we have concurrently developed two separate middleware systems for supporting application-level Quality-of-Service, namely, the $2K^Q$ and *Agilos* projects, with different design objectives and assumptions. The $2K^Q$ project [3] aims at QoS provisioning with the presence of OS-level resource reservation mechanisms, particularly focusing on two important aspects. First, it translates from application-level to 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 [2] 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 for 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 Fig. 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. As the graph becomes more complex, some services may contain application components that are clients as well.

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 case of $2K^Q$, 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. an application configuration), and subsequently reserve resources on each of the hosts in the component graph. In the case of *Agilos*, 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

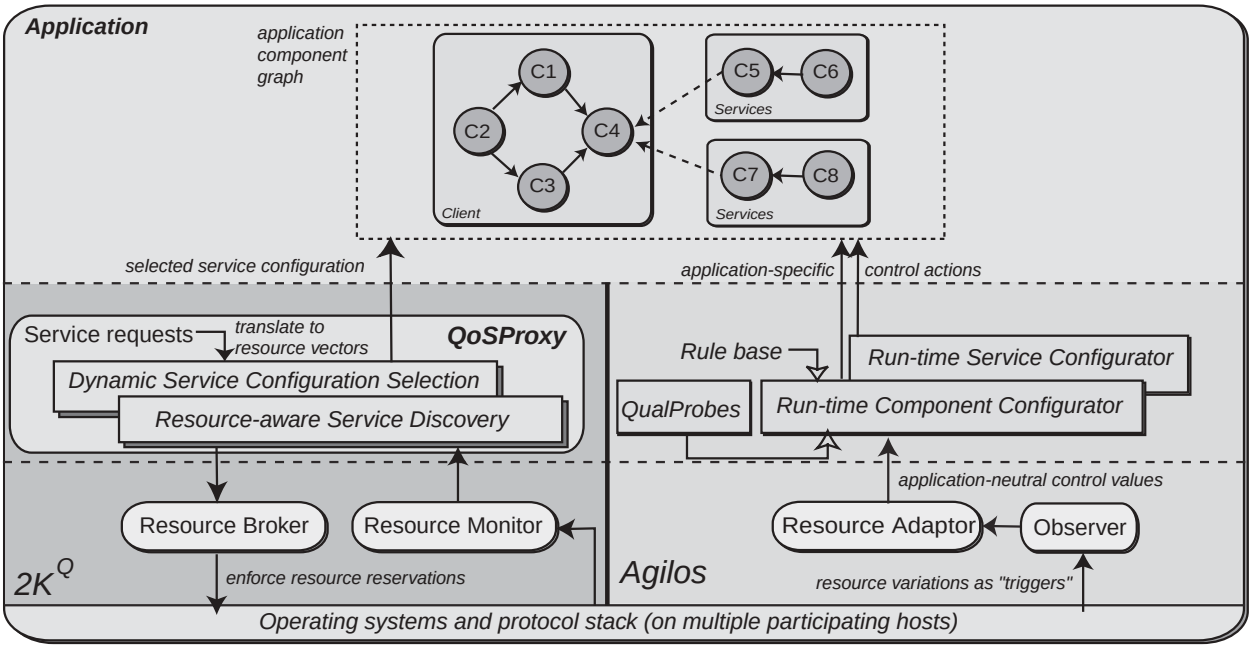

Fig. 1. A run-time comparison between the $2K^Q$ and *Agilos* middleware architectures

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 application configuration. Such resource checking is performed at application initialization time. The *Agilos* project, on the other hand, actively monitors resource usage via *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, to discover the mapping between application-level and resource-level QoS parameters, thus assisting the specification of the rule base.

Based on previous discussions, we further analyze the triggering sources and corresponding 'actions' of both middleware systems, with respect to their run-time support. Such analysis leads to the integration of these run-time solutions in a unified architecture, so that a complete and streamlined system can be designed.

3. Triggering sources of run-time middleware functions

Before we present our design of an integrated run-time support architecture, we first need to identify the triggering sources that activates such run-time support. We also note that this is not a 'cure-all' middleware solution for any applications. Instead, we consider a typical range of distributed multimedia applications such that they can be componentized and modeled with the application component graph. These applications include video streaming, video-over-IP telephony, video conferencing and visual tracking. They may be deployed over a variety of computing devices, from PDAs to workstation clusters. In such a dynamic and heterogeneous environment, it is necessary for the run-time support middleware to react to the following types of changes:

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 for multimedia streaming, or buffer space; (2) service-side parameters of similar types; and (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). Figure 2 illustrate an example, where the labels are shown as <parameter type, parameter value> pairs. 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.

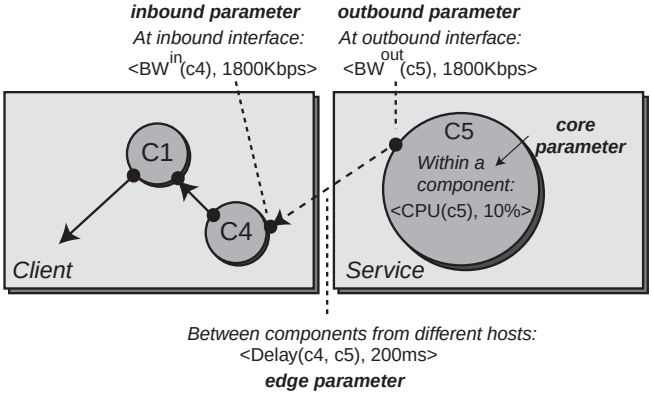

Fig. 2. Labeling the application component graph

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 video streaming clients) at different application QoS levels; secondly, 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. The user may move with its client host or between multiple client hosts, where he reconnects when he arrives at the new location (either wirelessly or with cables). In this case, 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'[1] of an application component to be one of its core parameters, it is observed that any user mobility may be treated as one of the following: (1) a *spontaneous reconfiguration* in the application component graph; or (2) 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 two categories of information: (I) the definition of the *application component graph*, including related parameters that are labeled in such a graph; (II) possible instances of alternative application configurations that may result from topological changes in the application component graph. These are represented by different topologies of the same pool of components. Any changes in such a data structure will activate run-time middleware functions; while any run-time function may also influence these application states. Such application states form the basis of discussion on our integrated architecture in the subsequent sections.

4. An integrated run-time support middleware architecture

We propose an integrated middleware architecture for run-time application support, providing the functions of run-time probing (monitoring) of application states, run-time instantiation of a specific service configuration, and run-time adaptation to application state variations. These three run-time support functions are driven by the same set of 'triggering sources' described in Sect. 3; while the application states are maintained on all participating end hosts in the distributed application. In this section, we present our design by detailing the *vertical* and *horizontal* aspects between middleware and application components involved: vertical aspects involve middleware and application components within the same host boundary; while horizontal aspects take place among middleware components across different hosts. We believe that, for effective monitoring and control of a complex distributed application, both vertical and horizontal aspects should be carefully designed in a streamlined fashion. The tenet of such a design is to provide simple, yet effective run-time support in probing, instantiating and adapting distributed applications.

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

4.1. Vertical aspects

Within the host boundary, we propose to represent the application states (defined in Sect. 3) as a database stored within each of the participating host. The application states is the focal point of operations to all the run-time middleware components, which include a *probe* component, a run-time *instantiation* component, and a run-time *adaptation* component. Each of these components is activated at run-time when certain triggering sources are detected based on the application states. By clearly defining the triggering sources and the activation timing of each of these components, we are able to operate the framework at run-time without ambiguity. We illustrate these components and their interactions in Fig. 3.

We describe the details of each component as in the following.

The probe.
For the purpose of QoS probing and monitoring of application states, the ***probe***, as a middleware component, should be activated periodically. When activated, it goes through two probing steps. First, it checks input-output relationships of current application components (within the same host boundary) against the recorded application states, which include an application component graph. If there exists a mismatch, it updates the application states to reflect the discovered changes. This step is necessary in order to detect topological changes in the application component graph within the host boundary. Second, it uses OS-level system calls or application-level hooks to measure all labeled parameters (including inbound, outbound, core and edge parameters) related to the components in the application states within the same host boundary. The probing results of the first two steps will be recorded in the database of application states. The probe is well aware of all system calls and application hooks necessary for monitoring any changes in resource and application parameters, as well as topology changes in the application component graph.

Run-time instantiation.
For the purpose of run-time instantiation, we design a 'processing engine' that makes application instantiation decisions solely based on the application states as input. The instantiation component is activated once at application start-up time. It collaborates horizontally with their counterparts on other involved hosts, to decide the actual application configuration under which the distributed application will be executed. Such a 'processing engine' utilizes the application configuration selection algorithm, the results of the algorithm (i.e. a particular application configuration among the possible alternatives) are enforced by the middleware. Since such enforcement updates the application component graph, it effectively changes the ap-

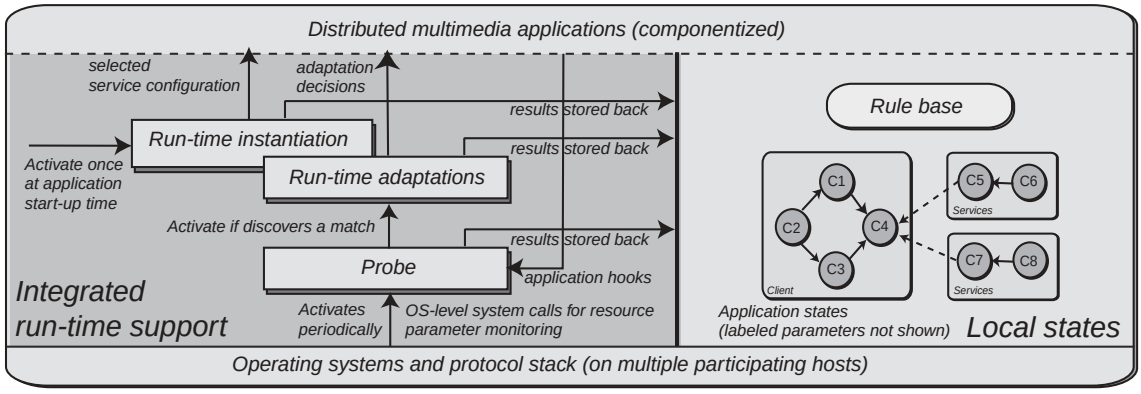

Fig. 3. An integrated run-time support architecture

plication states. After run-time instantiation, such changes should be stored back to the database of application states in each of the participating hosts.

The goal of the application configuration selection algorithm is to select an optimal application component graph (application configuration) that best reflects the current status with respect to all triggering sources. These include resource availability, QoS needs and user preferences in an application. The design of the algorithm was proposed in previous work [9] derived from our past experiences with the $2K^Q$ project, and we observe that it can be readily applied to this new integrated architecture.

Run-time adaptation.

For the purpose of run-time adaptations, we include a database of application-specific adaptation policies and rules (hereafter referred to as the *rule base*) based on the application states. Such a rule base is critical to the process of automating run-time adaptation that is tailored to the needs of a specific application. Assume the probe within the host boundary is activated on adjacent time instants t_1 and t_2, while the application component graph (monitored by the probe) on these time instants is G_{t_1} and G_{t_2}, respectively. A specific rule in the rule base is represented as a standard 'if ... then ...' clause, in the following form:

if $c_1(p_1(G_{t_1}), p_1(G_{t_2}), l_1, u_1)$
and $c_2(p_2(G_{t_1}), p_2(G_{t_2}), l_2, u_2)$
and ...
and $c_i(p_i(G_{t_1}), p_i(G_{t_2}), l_i, u_i)$
then *some adaptation action*

where each p_i may either be a labeled parameter, or the topology of the component graph. In the case of a parameter, $c_i(p_i(G_{t_1}), p_i(G_{t_2}), l_i, u_i)$ represents a desired condition that is dependent on the scalar value of this parameter at time t_1 and t_2, as well as a lower bound l_i and an upper bound u_i. If p_i is a certain topology of the component graph, then the condition c_i represents a condition that the topology satisfies, e.g. the removal of a certain edge in the graph at t_2 compared to the graph at t_1. Each p_i is considered as a triggering source to trigger adaptation controlled by the middleware component. In other words, adaptation takes place on satisfying the precondition of one of the rules in the rule base. Since the current states of each p_i are monitored by the probe, the correct behavior of run-time adaptation relies on the periodic activations of the probe. For the specific matching process to discover the set of rules to use, we reuse the inference engine based on fuzzy logic, presented in our previous work [2].

We further note that run-time adaptation – using a probing-decision-adaptation feedback loop – has the potential risk of leading to oscillations. This occurs when adaptation is too fine-grained and/or when probing results in an incorrect (inconsistent or inaccurate) image of the system/application state. There are previous work addressing this issue [10,11], while our previous work [2] analyzed the stability properties quantitatively by using classic control-theoretic approaches.

To summarize, our design is vertically simplified and streamlined compared with previous approaches. For example, all three components have clearly defined activation timing. While the instantiation component is activated only once at start-up time and the *probe* is activated periodically, the adaptation component is activated only when a match to the precondition of one of the rules is detected. Furthermore, all components take advantage of a unified database of application states. Whenever application states are modified, results of such changes are written back to the database. Since the triggering sources of all three aspects (probing, instantiating and adapting applications) are identical, this unified database makes it straightforward for the middleware to maintain current knowledge of the application in control.

In comparison to previous approaches, we note that various other middleware-based approaches (e.g. QuO) use complex multi-partite negotiations to establish a QoS contract. Such a contract is used in the subsequent request invocation or streaming phase as a reference for decisions concerning run-time adaptations, given observations on the actual performance during this phase. The streamlined design proposed in this paper does not require such complex negotiations, but uses a simple rule base to achieve similar results.

4.2. Horizontal aspects

Due to the distributed nature of applications, it is also critical to enable horizontal coordinations among peer middleware

components on multiple end hosts. This can be easily derived from the application component graph (Fig. 2), since only application components within the same host are grouped as one node in the top hierarchy, which consists of the topology between services and clients. On the other hand, the collection of the application states, most notably the application component graph, forms an entity requiring global information across the host boundary. For example, in Fig. 3, the probe of a single host can only update the labeled parameters that are attached to the components of the same host. Without horizontal interactions between probes on different hosts, some parameters in the application component graph may be out of date on any one of the hosts.

The above stated problem is caused by the logically centralized nature of application states for a certain distributed application. Assuming the existence of an omniscient observer that is able to collect all the changes from middleware components on all participating hosts, such an observer would be able to construct an accurate application component graph in a centralized fashion. Such an accurate 'snapshot' of the current component graph can thus be propagated to the databases of application states on all participating hosts. Without such an omniscient observer, the middleware components are responsible to interact in a peer-to-peer fashion, so that the application state changes are propagated in a timely and efficient way.

A trivial solution to this issue is that, whenever a change is detected in the application states (be it a parameter or topological change) on one of the hosts, such changes are propagated to all participating hosts in the application component graph. Obviously, a more efficient design is desired to reduce the overhead of propagating changes to the application states, with the trade-off of relaxed consistency across peer hosts. In the following, we present a *randomized propagation algorithm* to perform this task. The level of consistency is tunable in the algorithm to meet either more relaxed or more stringent requirements. This algorithm is implemented in the probe, on all participating hosts.

The randomized propagation algorithm

In this algorithm, the application component graph on each host is periodically propagated to a randomly selected subset of the participating hosts. The total number of hosts in this randomly selected subset, n, is a tunable parameter of the algorithm. Naturally, if n is equal to the total number of hosts (minus one), the algorithm upgrades to the standard solution (described previously), in which changes are propagated to all the participating hosts in the component graph. On the other hand, n is at least 1, i.e. changes are propagated to at least one of the hosts in the graph.

On receiving the propagated changes from other hosts, a host executes a **merge** algorithm to merge any new changes into its own application component graph. In order to distinguish new changes from outdated ones, a **version number** is labeled at each 'node' in the top hierarchy of the graph, where each 'node' represent either a client or a service. Figure 4 shows the results of attaching version numbers on the nodes representing clients and services.

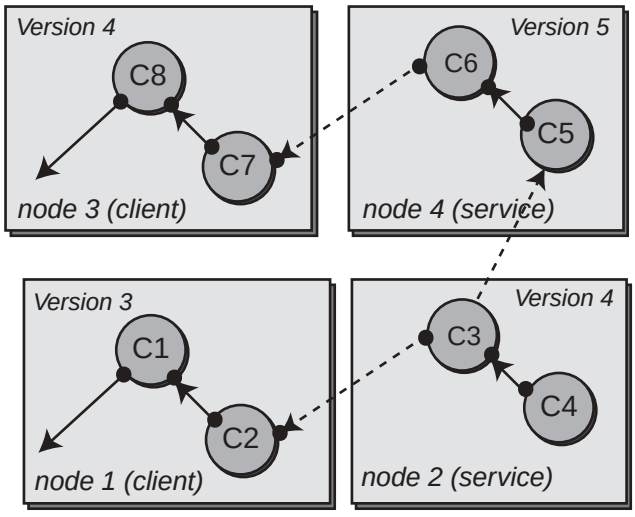

Fig. 4. Labeling version numbers in the application component graph

In this example, each node (either client or service) in the top hierarchy of the application component graph maintains a version number. The version number is incremented every time the host executes the randomized propagation algorithm. When a host in the selected subset receives such an update, it updates all changes (including parameter and topological changes) in the corresponding client or service node, if the local node has a smaller version number. The randomized propagation algorithm and the merge algorithm is presented in Tables 1 and 2, respectively. In these tables, we assume that each participating host (as well as its corresponding node, e.g. node 1 to 4 in Fig. 4, in the top hierarchy of the application component graph) has an identifier in the range of $(1 \ldots \mathbf{max})$, where **max** is the total number of participating hosts in the application.

Identical algorithms are used for constructing the initial application component graph on all participating hosts (or *bootstrapping*: each host propagates its own states (components on the same host, their relationships and parameters) to a selected subset of other hosts. It can be easily derived that, if there are no further changes caused by adaptation, the application component graph on any one of the participating hosts converges to accurate global states. In the case where adaptations on each host occur concurrently with propagations of states, there may be instances where the application component graphs on some of the hosts do not reflect the actual global states in the distributed application. Such inconsistencies may be eliminated by conservatively choosing $n = \mathbf{max} - 1$. However, we believe that a relaxed level of consistency is tolerable for the needs of run-time support, since adaptation only occurs when the application can not meet the minimum QoS needs; and the precondition of each adaptation rule only reflects such an extremely unfavorable condition. Therefore, if the propagation period T is tuned according to the frequency of minimum QoS violation occurrences (which can be estimated based on earlier entries in the application state databases), the relaxed level of consistency will not compromise the timeliness of adaptation actions taken by the run-time support middleware.

Table 1. The randomized propagation algorithm

The following algorithm is executed periodically with a period T:

If (changes exist in application component graph since last propagation)
Then
  For $i = 1$ to n (n: tunable parameter)
    m = random(1...**max**), s.t. m is neither previously selected nor the local host
    Propagate the local application component graph to host m
    If (changes exist in node m in top hierarchy of application component graph)
    Then
      $v(m) = v(m) + 1$, where $v(m)$ is version number of node m
  End
End

The following is executed when a topological change is activated for adaptation purposes:

For all nodes m involved in the topological change
  $v(m) = v(m) + 1$

Table 2. The merge algorithm

The following algorithm is executed on receiving an application component graph from a peer host m:

For $i = 1$ to **max**
  If $v_r(i) > v_l(i)$ (v_r: received version; v_l: local version)
  Then
    Copy node i *from* received application component graph
    to local application component graph
    $v_l(i) := v_r(i)$
  End
End

4.3. Implementation issues

With respect to implementation, one of the critical objectives of our design is to be able to implement the middleware framework without the need of a full-fledged middleware architecture, such as CORBA. This is in sharp contrast with existing proposals (e.g. reflective middleware, QuO), where the designed framework heavily relies on the existence of the CORBA architecture. This is made possible by minimizing the interface between application and middleware components for the purpose of monitoring and adapting applications.

Interface for probing. As may be observed from Fig. 3, one of the interfaces between applications and our run-time support architecture is the collection of application hooks that are necessary for the probe to accurately monitor application parameters, such as the frame rate in a multimedia streaming application. Although such application hooks may be implemented using a traditional middleware architecture such as CORBA, it is, obviously, not the only solution. Based on our experiments, the monitoring interface can be implemented in the form of a third-party application service (i.e. a daemon in UNIX or a service in Windows NT/2000). The applications may register and update their parameters within such a third-party service, while the probe queries the service asynchronously.

Interface for adaptations. The middleware components for run-time instantiations and adaptations also need to use an interface to instantiate and adapt the application. If the application is CORBA-aware (e.g. consisted of CORBA components), the middleware can take advantage of the defined interfaces of these application components to construct the component graph, or to tune the application parameters. However, even for mainstream applications without awareness of CORBA, we can still use a third-party service to accomplish this task. In this case, the middleware components update the parameters and component graphs registered within the third-party service, where the application queries the service periodically to adapt itself, taking the query results as 'hints' for the adaptation.

To summarize, our proposed run-time middleware support architecture does not need to rely on any existing middleware architectures such as CORBA. This brings substantial benefits towards the possibility of deploying the framework rapidly over existing platforms and applications. This is especially valuable given the current trend towards ubiquitous and pervasive computing environments, where it may not be feasible to deploy complex middleware architectures (e.g. CORBA) on some of the portable platforms (e.g. Pocket PC). Eliminating such dependencies on CORBA greatly improves the flexibility of the design to support a wide variety of applications on a wide variety of platforms. Indeed, based on our experiences, our framework can even be implemented as a library in binary form, that can be statically compiled or dynamically

loaded. Since the algorithms and inter-component interactions are generally streamlined, such a middleware layer poses minimum resource needs and restrictions in addition to those from the applications themselves.

5. An application case study with the unified framework

As a case study, we present and evaluate our experiences of implementing our proposed run-time support architecture to monitor and adapt a complex multimedia application, referred to as *online video streaming with object tracking*. The application is client/server based, with a server serving live feeds of video (from its attached camera) in the MPEG-2 format, and one or more clients receiving such live video feeds, tracking an interested moving object within the video. The clients may include heterogeneous portable devices running different OS platforms. This application is helpful to many real-world scenarios, such as live streaming of a football game to one or more clients with heterogeneous platforms, with one of the key players tracked.

Since the clients are on heterogeneous platforms (e.g. PalmOS vs. Windows) and have different levels of bandwidth or CPU availability (e.g. wireless PalmPilot vs. wired Pentium III-based notebook PC), there is a need for intermediate proxies to transcode the video to different formats, e.g. H.261 or uncompressed bitmap. The intermediate proxies are on separate hosts from the server and clients. Within the application, critical software components include the following: (1) The *MPEG* streaming service, located on the server; (2) The players that play video in either MPEG, H.261 or bitmap formats, depending on the bandwidth or CPU availability of each client; (3) The tracking filter that implements a collection of computationally intensive tracking algorithms, which processes incoming video and outputs the coordinates of the tracked object; and (4) The transcoders, which perform MPEG-to-H.261 or MPEG-to-bitmap transcoding. The application skeleton is illustrated in Fig. 5.

From the perspectives of probing, run-time instantiation and adaptation, we present examples of using our middleware components to provide run-time support to this example application. We will also show that, from our experiences, the streamlined design of our framework leads to straightforward implementations, keeping the modifications to the minimum on existing applications.

5.1. Probing

The *probe* monitors system-level and application-level parameters periodically. For system-level parameters such as inbound bandwidth to a host or CPU availability on a host, it uses OS system calls in UNIX and Performance Data Helper library in Windows NT. On starting the application, the probe is started automatically (as a dynamically shared library) and is activated periodically within a different thread. Probes on all participating hosts store probing results in the local application component graph. Thanks to the randomized propagation algorithm, our experiences show that each of the local component graphs converge to reflect accurate global states after an initial period of 'stabilization' time. For faster convergence, we set $n = \mathbf{max} - 1$ in the randomized propagation algorithm during this stabilization period.

5.2. Run-time instantiation

During the stabilization period, the probe on each end host checks its local resource availability, and propagates the results to its peers on other end hosts. When every probe component has collected the global resource availability information, it will pass the information to the instantiation component on the same end host. The latter will treat the information as the initial triggering source, and therefore starts the key function of run-time application instantiation.

More specifically, run-time instantiation involves the selection of an appropriate configuration for the application, so that it will execute properly under the specific resource availability condition. There are four possible configurations for our example application, each of which is suitable for a different resource availability condition:

- Configuration C_1 involves three application components: MPEG streaming service on the server, tracking filter and MPEG player on the client. C_1 is suitable for a high performance client which has sufficient CPU, bandwidth, and energy resources.
- Configuration C_2 involves four application components: MPEG streaming service on the server, MPEG-to-H.261 transcoder on the proxy, and tracking filter and H.261 player on the client. C_2 is suitable for a client with sufficient CPU and energy resources, but without a high-speed network connection.
- Configuration C_3 consists of four application components: MPEG streaming service and tracking filter on the server, MPEG-to-bitmap transcoder on the proxy, and Bitmap player on the client. C_3 targets the following situation: the client is weak in its CPU, bandwidth, and energy resources (for example, a handheld PC); and the server has higher CPU capacity than the proxy. Since the client is weak, Bitmap player is used to avoid CPU and energy consuming decoding[2]. In addition, the tracking filter is off-loaded from the client to the server.
- Configuration C_4 involves the same set of application components as configuration C_3. However, the tracking filter is now on the proxy instead of on the server. C_4 also helps a weak client. However, it is a better choice than C_3 when the proxy has higher CPU capacity than the server.

The selection among C_1, C_2, C_3, and C_4 is governed by a rule base in the following form (for simplicity, the triggering source involves an incomplete set of resources):

if $\text{CPU}_{\text{server}} \in [20\%, 100\%]$
and $\text{BW}^{\text{in}}_{\text{client}} \in [1Mbps, \infty]$
and $\text{ENERGY}_{\text{client}} \in [100\%, 100\%]$
then select C_1

if $\text{CPU}_{\text{server}} \in [20\%, 100\%]$

[2] We assume that the MPEG-to-bitmap transcoding will significantly degrade the media data rate and quality.

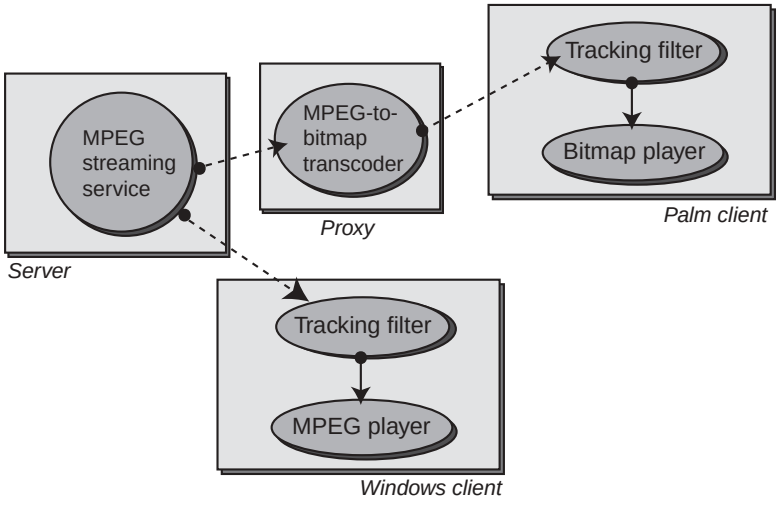

Fig. 5. Case study: online video streaming with object tracking

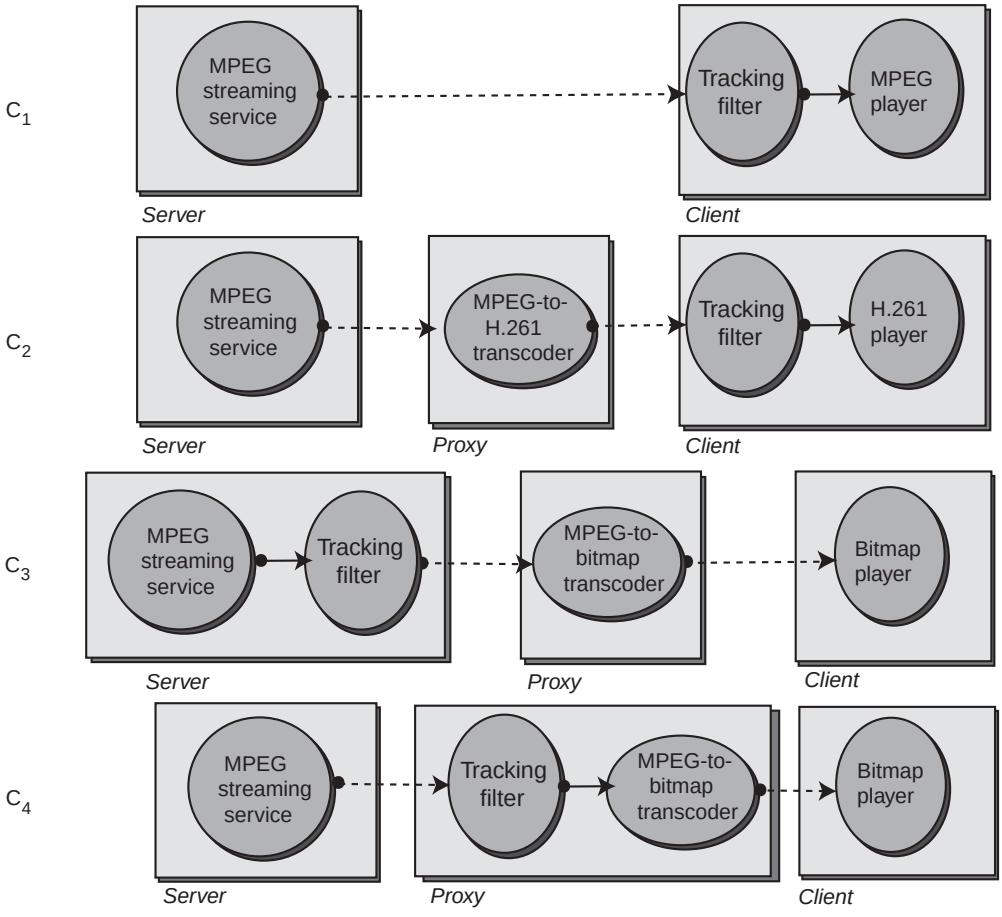

Fig. 6. Possible application configurations: C_1 to C_4

and $\text{CPU}_{\text{proxy}} \in [20\%, 100\%]$
and $\text{BW}^{\text{in}}_{\text{client}} \in [64Kbps, 1Mbps)$
and $\text{ENERGY}_{\text{client}} \in [100\%, 100\%]$
then select C_2

if $\text{CPU}_{\text{server}} \in [40\%, 100\%]$
and $\text{CPU}_{\text{proxy}} \in [20\%, 40\%)$
and $\text{BW}^{\text{in}}_{\text{client}} \in [16Kbps, 64Kbps)$
and $\text{ENERGY}_{\text{client}} \in [20\%, 40\%]$ **then** select C_3

if $\text{CPU}_{\text{server}} \in [20\%, 40\%)$
and $\text{CPU}_{\text{proxy}} \in [40\%, 100\%]$
and $\text{BW}^{\text{in}}_{\text{client}} \in [16Kbps, 64Kbps)$
and $\text{ENERGY}_{\text{client}} \in [20\%, 40\%]$
then select C_4

Every instantiation component has a copy of the rule base above. To select the appropriate application configuration, an instantiation component compares the global resource availability information against the precondition of each rule in the rule base. For example, if the global resource availability shows that the client's inbound bandwidth is 32Kbps; and the server CPU capacity (60%) is larger than the proxy CPU capacity (20%), then configuration C_3 will be selected. We note that the selection may be made by the instantiation component on *any* end host, which will then propagate the selection to its peers on other end hosts. Even if selections are made concurrently on different hosts, the selections should be identical. This is because the global resource availability information has been propagated to each end host during probing, and the same rule base is used on each end host.

After the application configuration has been selected and the selection is known by every end host, the application components involved in the selected configuration will be selected and activated by the corresponding instantiation component. Similarly, those not involved in the selected configuration will be deactivated. Meanwhile, the application state database on each end host will be updated, including the application component graph (which represents the selected configuration) and the corresponding label values on the graph. During the execution, the application state database will be periodically updated by the *probe*.

5.3. Run-time adaptation

Run-time instantiation is activated only once at application start-up time (after an initial stabilization period). In contrast, run-time adaptation is activated only when any one of the adaptation rules in the rule base is satisfied. As an example, one of the rules in the rule base is in the following form:

if $\text{CPU}_{\text{server}} < 10\%$
and $\text{CPU}_{\text{proxy}} > 80\%$
and C_3 is selected
then select C_4

Such a rule switches the application configuration from C_3 to C_4 if original justifications to select C_3 no longer hold; rather, the opposite situation is detected by the probe. Obviously, the sensitivity (responsiveness) of adaptation depends on the actual rules used [2].

5.4. Implementation discussions

Because the objective of our design is *simplicity*, we were able to deploy the middleware framework to monitor and adapt the video streaming and tracking application in a very short period of time. We have also attempted to use the previous $2K^Q$ and *Agilos* architectures to instantiate and adapt the same application. We have observed that, in our proposed framework without the support of CORBA, the results of instantiation and adaptation behavior show no differences compared with previous results. If we consider the fact that both $2K^Q$ and *Agilos* rely on CORBA to function correctly, the verdict is that considerable implementation efforts can be saved using our new, streamlined middleware design. This is especially true if the application is not CORBA-aware, since modifications to a typical application so that it becomes aware of CORBA is not a trivial task. As an additional note, although we generally use $n = \textbf{max} - 1$ in the randomized propagation algorithm, we have not detected any unexpected adaptation behavior when $n <$ **max** - 1. This may be due to the limitation of using a local area network environment to perform all experiments.

6. Conclusion

In this paper, we have presented a unified middleware architecture to integrate various run-time solutions found in previous middleware designs with different perspectives. 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 past experience in 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 platforms. 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. With a case study involving a complex multimedia application, we have demonstrated that the implementation of such a streamlined, integrated run-time framework effectively controls and monitors the application behavior. The main objective of this paper is to discuss and present the merits involved in such a streamlined design. As part of our future work, we plan to study the effects of our middleware framework on more applications, in order to further simplify our design as much as possible.

Acknowledgements. The valuable and insightful comments from the anonymous reviewers are much appreciated. They were helpful in the process of revising and improving the quality of this paper.

References

1. Chang F, Karamcheti V (2001) A Framework for Automatic Adaptation of Tunable Distributed Applications. In: Cluster Computing: The Journal of Networks, Software, and Applications, Volume 4, Number 1, 2001.
2. Li B, Nahrstedt K (1999) A control-based middleware framework for Quality of Service adaptations. IEEE J Sel Areas Commun (Special Issue on Service Enabling Platforms) 17(9):1632–1650
3. Wichadakul D, Nahrstedt K, Gu X, Xu D (2001) 2KQ+: An integrated approach of QoS compilation and component-based, run-time middleware for the unified QoS management framework. In: Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001). Heidelberg, Germany, November 2001
4. Zinky J, Bakken D, Schantz R (1997) Architectural support for Quality of Service for CORBA objects, Theory and practice of object systems. A Wiley journal publication

5. Stiller B, Class C, Waldvogel M, Caronni G, Bauer D (1999) A flexible middleware for multimedia communication: design, implementation, and experience. IEEE J Sel Areas Commun 17:1580–1598
6. Fitzpatrick T, Blair G, Coulson G, Davies N, Robin P (1998) Supporting adaptive multimedia applications through open bindings. In: Proceedings of International Conference on Configurable Distributed Systems (ICCDS '98). Annaolis, Maryland, pp. 128–135
7. Chandra P, Fisher A, Kosak C, Ng T, Steenkiste P, Takahashi E, Zhang H (1998) Darwin: resource management for value-added customizable network service. In: Proceedings of IEEE International Conference on Network Protocols (ICNP '98). Austin, Texas, pp. 177-188
8. Coulson G, Costa F, Duran H (2000) On the design of reflective middleware platforms. In: Proceedings of Workshop on Reflective Middleware (RM 2000). New York, USA
9. Xu D, Wichadakul D, Nahrstedt K (2000) Multimedia service configuration and reservation in heterogeneous environments. In: Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS 00). Taipei, Taiwan, pp. 512–521
10. Angin O, Campbell A, Kounavis M, Liao R (1998) The Mobiware Toolkit: programmable support for adaptive mobile networking. IEEE Personal Commun Mag **5**(4):32–43
11. Noble B, Satyanarayanan M, Narayanan D, Tilton J, Flinn J, Walker K (1997) Agile application-aware adaptation for mobility. In: Proceedings of the 16th ACM Symposium on Operating System Principles. Saint-Malo, France, pp. 276–287