The Design and Implementation of Service Reservations in Real-Time SOA

Mark Panahi, Weiran Nie, Kwei-Jay Lin
Department of Electrical Engineering and Computer Science
University of California, Irvine
Irvine, CA 92697-2625, USA

Abstract—Service-oriented architecture (SOA) provides the flexibility of dynamically composing business processes in enterprise computing. However, they must be enhanced to support real-time activities in future SOA applications such as media streaming, control applications, cyber-physical, and intelligent vehicle systems. In this paper, we present the RT-Llama SOA middleware to support predictability in real-time processes. Given a user-specified service process and deadline, the RT-Llama resource management system will reserve resources in advance for each service in the process to ensure the process can meet its end-to-end deadline. RT-Llama provides a global resource manager (GRM) to compose and reserve services in a business process. The GRM is facilitated by an efficient data structure that keeps track of the utilization of local resources. The data structure, called \textit{dTBTree}, is updated by each local resource manager (LRM) periodically and cached in the Host Utilization Repository in GRM. We have implemented RT-Llama's resource management components so that they can efficiently adapt to dynamic real-time environments.

I. INTRODUCTION

Real-time service-oriented architecture (RT-SOA) is a new and challenging research area. As SOA gains more acceptance and becomes the prevailing software paradigm for dynamically integrating loosely-coupled services into one cohesive business process (BP) [1], [2], the confluence of real-time and SOA systems is inevitable. We must prepare SOA for meeting the predictability requirements of real-time enterprise systems, event-driven architectures, and sensor-based applications.

Current SOA solutions have not addressed the strict predictability demands that many applications require, from banking and finance to industrial automation and manufacturing. Such applications, many of whom already embrace SOA for a large part of their architecture, would greatly benefit from a comprehensive SOA solution that can also encompass their event-driven and real-time components.

While some aspects of SOA make its transition to real-time simpler, still other aspects pose serious challenges. SOA-based systems may dynamically integrate services, created by either enterprises internally or external service providers. Such flexibility allows an SOA-based system to dynamically reconfigure a business process, even at runtime, in order to meet timing constraints. On the other hand, the uncertainty of the execution environment due to dynamic service binding in SOA makes it very difficult to provide any form of performance guarantee before runtime. We need to design a service reservation scheme so that a BP can have a guarantee on the system resources it needs to deliver real-time predictability.

IT companies have been developing event-driven system technology for many years. Some have produced products (e.g. Microsoft Dynamics, IBM WebSphere Business Events, SAP Smart Items, etc.) to interconnect and manage real-time data flow in enterprise systems. However, very few projects have studied the issue of service reservation that is essential to ensure a predictable BP execution.

In this paper, we present the RT-Llama project (as an extension of Llama [3], [4]) which meets the real-time enterprise challenge by enabling SOA users to pre-schedule an entire BP, thus eliminating the risk of missed deadlines due to the over-utilization of resources. RT-Llama differs from previous service-oriented architectures in that it allows end-to-end BP deadline guarantees through advance reservations of local resources.

In order to make this work, we 1) create global resource management and composition components that reserve resources in advance for each service in a BP to ensure it meets its end-to-end deadline; 2) leverage a very efficient data structure for managing reservation utilization data, a dynamic adaptation of the TBTree [5] called the \textit{dTBTree}; 3) develop a pre-screening mechanism that utilizes cached versions of the dTBTree to increase the likelihood of successful reservations; and 4) create a CPU bandwidth management system for each host essentially dividing a CPU into multiple temporally-isolated virtual CPUs, that can be mapped to a higher-level service model allowing providers to create different classes of service with various levels of predictability.

We evaluate the pre-screening mechanism under different workload scenarios and show how varying the cached dTBTree’s height and update frequency can effectively manage...
the tradeoffs between resource overhead and reservation success.

The rest of the paper is organized as follows. Sec. II reviews the challenges of bringing real-time to SOA. Sec. III presents the RT-Llama RT-SOA architecture. We present the performance study of the RT-Llama implementation in Sec. IV. Related work is compared in Sec. V.

II. BACKGROUND

A. Scope of Real Time Support

Supporting predictability in SOA systems introduces a broad array of challenges. When trying to reserve distributed services, several aspects need to be considered, including service availability, the types of physical resources required, and the structure of the business process (BP). Here, we discuss the scope of efforts to bring predictability to SOA – specifically, what challenges we have addressed in this work, and the challenges that still remain ahead of us.

- **BP Scheduling.** Typically, BPs are composed and then reused (executed more than once) by creating multiple process instances. However, since RT-Llama’s composition process takes into account changing service attributes and availability, reusing BPs becomes challenging. We foresee two major ways of scheduling BPs. First, our current study is that of a “one-shot” (single instance per BP) model for BP composition and execution. Even if a user has the same requirements across executions, the BP may be regenerated, possibly selecting different services for each BP. Second, we also plan to provide multi-instance BP scheduling, which can include either periodic or sporadic schemes. The challenge with this approach is to ensure a service’s availability across multiple BPs executions during the BP service selection and reservation phase.

- **Reservation Timeliness Model.** The RT-Llama architecture is based on the advance reservations of service executions to guarantee timeliness. There are two ways to carry out a reservation procedure depending on how far in advance the BP may be scheduled. In this paper, we assume that reservation requests are made well enough in advance that they will not impinge on the BP instance start time. As future work, we plan on making RT-Llama more flexible and allow “just-in-time” reservations. In that way, reservation requests themselves are subject to real-time requirements.

- **BP Structural Model.** BPs may vary in complexity, from simple sequential BPs to more intricate ones that include many branch-merge points and parallel segments. Our current study addresses sequential pipeline BPs. However, we would like to explore the impact of more complex BPs on schedulability. For instance, in a branch-merge case, we will necessarily have to reserve services that will never be used because the branch conditions are unknown at reservation-time. Moreover, when dealing with parallel segments, synchronization at the end of such segments would be required.

- **Types of Resources.** In SOA systems, and distributed applications more generally, there are a variety of resources that can be a source of contention and thus introduce unpredictability. In this paper, we study the predictability of CPU resources. However, not all BPs may be composed of services that are completely CPU-driven. Other varieties of services may be data-driven creating contention at the hard disk, while others face restrictions at the network level. We will address both data and network resources in future work.

- **Adoption of BPEL.** Some real-time systems determine admissibility of a task once it arrives at the host (i.e., admission control). For RT-SOA applications, this may not be acceptable. If tight predictability limits are required for a BP, it would take only one service’s admission rejection or cost overrun to severely disrupt the whole BP’s execution. Using SOA, the entire BP is usually encoded in a higher level language, such as BPEL. The execution path is planned before hand and can be reserved in advance. Therefore, we leverage this information in our RT-Llama framework and build advance reservation mechanisms for the process so that the risk of service rejections and cost overruns is eliminated.

In order to ensure end-to-end predictability for any real-time distributed system, every subcomponent or system layer must also provide predictability. These include the operating system, communication infrastructure, BP composition infrastructure, distribution middleware, and client infrastructure. In our current work, we focus on introducing predictability into two of the main layers mentioned above: **BP composition infrastructure** and **distribution middleware.** We leverage Real-time Java [6] and the Solaris 10 operating system to provide real-time scheduling capabilities to our middleware.

B. Service Model

In contrast to some existing distribution frameworks, SOA can support multiple classes of a service that are offered by providers at different price points. In RT-Llama, providers can deploy services under two categories:

- **Unreserved:** Services deployed under this category accept requests without any prior workload reservation. There are two main classes:
  - **Immediate** requests are serviced in first-in first-out (FIFO) order according to the operating system’s underlying real-time scheduler. There may be at
most one immediate class. However, it may support several priorities.

- **Background** requests are serviced in a best effort fashion according to the operating system’s underlying non-real-time scheduler.

- **Reserved**: Services deployed under this category may only receive requests that have been reserved in advance. There may be any number of reserved classes. Reserved classes are intended to map to different levels of service that providers would like to offer.

### C. Real Time Model

We define the real-time task model for SOA as follows: A business process $BP_i$ is a workflow composed of a sequence of service invocations $S_{i,j}$. Each $BP_i$ begins execution at time $r_{i,j}$, finishes at time $f_{i,j}$, with an execution time of $c_{i,j}$, and has a deadline of $d_{i,j}$ that is respected if $f_{i,j} \leq d_{i,j}$. During execution, each service invocation arrives at time $r_{i,j}$ with a deadline of $d_{i,j}$, finishes at time $f_{i,j}$ after executing for $c_{i,j}$.

Since the RT-Llama framework supports advance reservations, we have $a_{i,j}$ as the scheduled start time for $BP_i$, and $a_{i,j}$ as the scheduled start time for $S_{i,j}$. We thus have the relation $w_{i,j} = d_{i,j} - a_{i,j}$ as the time window for $BP_i$ and $w_{i,j} = d_{i,j} - a_{i,j}$ as the time window for $S_{i,j}$. Moreover, each $S_{i,j}$ is assumed to have a worst case execution time $wC_{e_{i,j}}$.

This system model must be supported by the underlying infrastructure. In our prototype environment, each host that deploys this framework has at least 2 CPU’s or a dual core processor, with one CPU/core completely devoted to servicing real-time tasks (RT-CPU) and the other available for background tasks and other operating system tasks (non-RT-CPU). The RT-CPU will execute both the unreserved immediate class, as well as all reserved classes. These classes, although sharing the same CPU, must be **temporally isolated** from each other, meaning that cost overruns that may occur in one class cannot interfere with other classes. This is accomplished by creating multiple virtual CPUs out of the RT-CPU. Therefore, each class is assigned to a virtual CPU, which in turn has a system defined bandwidth percentage. This is discussed in detail in Sec. III-C.

### III. Architecture

#### A. RT-Llama Architecture

RT-Llama is comprised of several components that aid providers in offering predictable service execution and users in ensuring that their BPs meet their specified deadlines [7]. The main architecture is shown in Fig. 1. From a user’s perspective, all that is required of them is to specify input values, desired output, and timeliness parameters, including start time and deadline. Based on this information, RT-Llama selects and reserves a feasible BP that matches the user’s requirements.

The **Global Resource Manager (GRM)** in RT-Llama is responsible for scheduling a user’s BP based on their requirements. It has the following subcomponents.

- **QBroker/RT-QBroker** is a QoS broker designed to select a BP based on user specified constraints, and has been previously studied in [8]. In addition to its original role for best-effort BPs, RT-QBroker has been designed to perform feasibility checks on individual services during service selection, by consulting the Host Utilization Repository, to determine if the host is likely to be available during the general span of time of the BP.

- **The Business Process Scheduler (BPS)** is responsible for selecting a BP composed by RT-QBroker according to either a concurrent or sequential mechanism (discussed in [7]) by contacting the host of each service. In the event of a reservation failure (i.e., the reservation is not feasible or the request timeout is exceeded) the BPS requests a new BP from RT-QBroker, absent any unfeasible services. If RT-QBroker cannot find a feasible BP, the user’s request is rejected.

- **The Host Utilization Repository (HUR)** stores cached future utilization information regarding each host using dTBTrees. As a cache, it may be updated at predefined intervals, and thus not always completely up-to-date. It may also have coarser grain information than that stored on the host to reduce overhead. However, it provides a quick way of determining if a service is likely to be available at a future time.

The **Local Resource Manager (LRM)** is responsible for hosting services and ensuring that requests on such services are executed predictably by working with the GRM components. It has the following subcomponents.

- **The Reservation Manager (RM)** manages advance reservations for the host. It keeps track of the reserved classes to manage their overall utilization using dTB-Trees (consulted during the planning stage), as well as a simple hashmap to manage detailed reservation information (consulted during the execution stage). Both data structures are represented by the Reservation Repository (RR). Additionally, it sends asynchronous updates on recent utilization changes to the HUR.

- **The Admission Controller (AC)** accepts incoming service requests and routes them to an **Executor** responsible for servicing requests for the various classes of service. Each executor is mapped to a virtual CPU (discussed in Sec. III-C) on the underlying operating system. The real-time executors include the **Unreserved Executor** and the **Reserved Executors** and are each...
given a pre-defined share of bandwidth on the RT-CPU. For reserved requests, it looks up $a_{i,j}$, $w_{i,j}$, and $d_{i,j}$ according to the request’s reservation id in the RR. This information is necessary for the Reserved Executors to ensure that requests are service in an earliest deadline first (EDF) fashion and to properly accommodate tardy requests and deadline overruns. The Best-effort Executor, by contrast, executes on the non-RT-CPU along with other system threads and provides no guarantees for predictability.

B. Reservation Data Management

One of the unique features of the RT-Llama framework in contrast to other SOA systems is its ability to reserve service requests in advance. This avoids the pessimistic nature of on-demand admission control strategies and leads to potentially higher utilization rates as users are able to plan their BP executions ahead-of-time.

In the RT-Llama framework, we select the temporal bin tree (TBTree) solution discussed in [5] due to its flexibility, efficiency in both time and space, and suitability for real-time planning applications. The TBTree uses a binary tree structure to store the total amount of time available within an interval at different levels of granularity. Each node in the tree contains the sum of the available time in the left and right children, each of which covering half the time interval of the parent. A search for available time begins at the root and descends into the tree and identifies the leaf node that can completely accommodate the task, specified as either an earliest start time or latest finish time, and the expected execution time. For improved efficiency, the TBTree can implement forward pointers for each node, which indicate the next part of the tree to search in the event that the given node cannot accommodate the task, thereby avoiding costly backtracking operations.

This data structure was originally designed in [5] to store advanced reservations for fixed but long time horizons in manufacturing systems where tasks may be planned out days or weeks in advance and take minutes or hours. By contrast, we adapt this data structure to the RT-SOA environment since we expect service executions to be planned out minutes (occasionally, tens of seconds) in advance. Moreover, we expect individual service execution times to be also on the order of seconds.

These new requirements impel us to make the following two adaptations to the TBTree: 1) small task granularities and 2) subtree recycling. We call this new data structure the dynamic TBTree, or dTBTree.

1) dTBTree Leaf-node Granularity: In the design of the original TBTree, the capacity of each leaf node is greater or equal to the largest possible task size. Moreover, any task must be completely accommodated within a leaf node. In other words, tasks cannot span across multiple leaf nodes. However, the problem with this approach in its application to RT-SOA is that it leads to underutilization of resources. Instead, we make the capacity of each leaf node as the smallest possible task size and allow tasks to span across multiple leaf nodes. During the search operation, the dTBTree’s forward pointers are utilized to laterally traverse the tree and available time slots are collected in a list. Since the HUR is used only to guide the reservation pre-screening process, the cached versions of the local dTBTrees may have coarser granularities to manage space/accuracy tradeoffs.

Fig. 1. RT-Llama architecture.
2) dTBTree Subtree Recycling: Having a long time horizon with small task granularities would require an prohibitively large TBTree. Therefore, our second change is to enhance the original TBTree with dynamic capabilities to recycle subtrees whose right-most leaf node’s time index is less than the current system time, called subtree expiration.

Fig. 2 depicts a dTBTree and how it operates. The left and right subtrees are swapped since the left subtree has expired. Each node has two fields: (subtree id, remaining capacity). The dTBTree offers two recycling policies: 1) eager reinitialization where the capacities of each of the nodes of the expired subtree are reinitialized during the recycling operation itself; and 2) lazy reinitialization where the node capacities are only reinitialized during search operations. For the lazy reinitialization approach, while descending into the tree during a search operation, the subtree id is compared to that of its parent. If they are different, the subtree id is set to the parent’s subtree id, and the time capacity is reset to the maximum (calculated based on the node’s depth).

During a subtree recycling operation, several things need to happen; 1) the forward pointers along the right side of the non-expired subtree need to be set to point to the root of the recycled subtree, 2) the used capacity of the recycled subtree needs to be added back to the root, 3) the root’s pointers to the left and right must be swapped, 4) for the root node of the recycled subtree the subtree id must be given a new unique value and its capacity must be replenished. These steps are shown in phase 2 of Fig. 2 where the recycling operation commences at time 4. The search operation is depicted in phase 3.

If we compare the running time of the lazy versus the eager approach, we see that with the lazy approach, the only non-constant-time operation is step 1, setting the forward pointers of the non-expired subtree, which will run in logarithmic time to the size of a subtree. The eager approach, on the other hand, will require linear time to reinitialize the subtree. The lazy approach will become especially important for future work when we explore adding real-time constraints on the reservation system itself (i.e., “just-in-time reservations”).

3) dTBTree Usage in RT-Llama: The dTBTree is used in the LRM to store utilization information for each reserved executor. Moreover, there is a cached version of each local dTBTree in the HUR of the GRM. This cached dTBTree is an approximation of the original in both time and space in that it receives change updates only after specified intervals, and can have a height that is shorter than that of the original, if space is a constraint and can be traded for accuracy. In Sec. IV, we will study the impact of varying these two parameters on the effectiveness of the pre-screening process.

In terms of using a dTBTree for real systems, a typical scenario would involve creating a tree to accommodate tasks as far ahead as 1 hour, with 100 ms leaf node granularities.

We could create a tree of height 16 with 65,536 leaf nodes and 131,071 total nodes in the tree, yielding 1.82 hours of reservation time. These figures would be doubled in order to accommodate subtree recycling. In order to optimize on space, the subtree recycling procedure could be designed to recycle smaller parts of the tree at a time (e.g., a quarter or eighth, instead of half); this is to be explored in the future.

C. CPU Bandwidth Management

In order to create a framework that enables users to choose different levels of service, RT-Llama uses a method for mapping such levels of service to temporally isolated virtual CPUs (VCUs) that can be assigned bandwidth percentages of a physical CPU. Our VCPU mechanism is inspired by the constant bandwidth server (CBS) CPU scheduling mechanism [9].

There are two workers for each VCPU: a work-conserving worker (WCW) that services any requests that arrive at least T time units before their start time, and a higher priority current job worker (CJW) for requests that arrive within T time units of their start time. T is a configurable parameter that is currently set to the largest worst case execution time (WCET) for any service in the host. Each worker has its own queue. The advantage of this dual-worker approach is to ensure that the system is as work conserving as possible (i.e.,
a VCPU is never idle if there is work to be done) but makes sure the current tasks are always executed immediately (i.e., not delayed by a currently executing task that has arrived well ahead of its start time).

When a service request arrives at an LRM, the admission controller (AC) reads the reservation id from the request’s HTTP header. The AC consults the Reservation Manager (RM) to determine the request’s reservation information, i.e., priority class, start time, WCET, and deadline. The AC then determines, based on the value of T and the request’s start time, whether to send the request to WCW queue or the CJW queue for the given VCPU.

Bandwidth shares are implemented by rotating the priorities of each VCPU’s WCW and CJW. A VCPU is activated by giving it the highest priority relative to the other VCPUs for a certain length of time $V_i$ out of a period $P$ such that $V_i/P$ is the VCPU’s bandwidth percentage. Within each VCPU, the CJW is given a higher priority than the WCW. For example, if class H is assigned 60%, class L is assigned 40%, and a period is 100 ms, then for 60 ms H’s CJW and WCW will be given priorities $p$ and $p-1$, respectively. L’s CJW and WCW will be given priorities $p-2$ and $p-3$, respectively. For the remaining 40ms, L will be given $p$ and $p-1$, H will be given $p-2$ and $p-3$. RT-Llama’s bandwidth management component is responsible for managing such priorities. It executes at the highest system priority, assigns VCPU priorities and then sleeps $V_i$ to allow execution to proceed, after which it again reassigns priorities.

RT-Llama’s VCPU scheme is initially implemented using RT-Java on top of Solaris 10 installed on a 2-CPU, 1.34-GHz UltraSPARC/1-Mbyte L2 cache/512 Mbytes RAM rackmount server. We have observed that VCPU bandwidth management overhead is low with each reassignment procedure requiring at most 500 microseconds. We plan to port the implementation to other popular servers in the near future and conduct more testing on its performance.

IV. Performance Study

We have conducted simulations to test the effectiveness of the GRM pre-screening function. One unique feature in the RT-Llama architecture is to use the cached host utilization in HUR as a hint for real-time business process (BP) composition in order to increase reservation efficiency. To focus on the real-time requirement of a BP rather than the complexity of the BP structure, all BPs tested have only sequential flow structure.

We have tested the following system parameters in our study.

1) pre-screening threshold. As mentioned before, the GRM avoids selecting services with a very high host utilization because of their likelihood to miss deadlines. Specifically, we set up a utilization threshold $k$. Whenever a service is to be selected in a business process, we require that its host utilization be under a threshold $k$. As a point of reference, a $k$ value of 1 represents no pre-screening. If all candidate services cannot satisfy the condition, the GRM will notify the user about the reservation failure without actually performing the reservation on LRM.

2) workload. We generate a certain amount of workload for the simulation. A workload factor of 1.0 represents a workload exactly equal to the full system capacity, which is the maximum amount of work that can be handled. A workload factor of 1.5 represents a workload 150% of the system capacity. In general, we are more interested in systems with overloads.

3) dTBTree update. Caching utilization information in GRM requires constant dTBTree updates. A system may adjust the update overhead by changing the update interval and tree height. A shorter interval creates more frequent system update overhead, and a larger tree height incurs more data to be cached.

We then record the following system performance attributes in the simulation.

1) success ratio: the number of successful reservation requests divided by the total number of reservation requests. This parameter is defined from a user’s perspective and it does not distinguish between whether the reservation attempt was stopped at the pre-screening stage or by the LRM itself.

2) efficiency: the number of successful reservation requests divided by the number of reservation requests that passes pre-screening. Efficiency, in contrast with the success ratio, measures the success of reservations that only pass the pre-screening process, and thus captures the overhead spared by avoiding remote reservation operations.

In the following experiments, we treat the host as a single reserved executor (Section III-A) for simplicity. However, our simulation applies to multiple reserved executors, as they are temporally isolated from each other. We simulate 5 hosts. A dTBTree with 1024 time units is associated with each host to record the utilization information. Services with the same function are replicated on each of the 5 hosts so that we can always switch to a underloaded host if there is one. In the simulation, we use a process containing 3 services with each service having an execution time of 1 time unit. Process instances differ in their start times and have deadlines varying between 20 and 32 time units. To simulate a more realistic environment, we pre-loaded the system with 30% of system capacity before beginning to collect data.

We have designed experiments to see the effect of host
threshold $k$, workload factor, and HUR update interval on the RT-QBroker pre-screening success ratio and efficiency. Since our goal is to jointly optimize both the success ratio and efficiency, we also plot the sum of these two performance values as a general indicator of the joint optimum. The results of our experiments are shown in Figures 3 and 4. Choosing the optimization criteria to be that of maximizing the sum of these two values, we see that when the workload factor is 1, the optimal $k$ is around 0.85. However, when the workload factor increases to 1.5, the optimal $k$ decreases to about 0.6. The implications for real systems is that the $k$ value should be varied according to historical or expected workload to achieve the optimal level of reservation success ratio and efficiency.

Fig. 3. Pre-screening result for (workload = 1.0 and HUR interval = 2)

Fig. 4. Pre-screening result for (workload = 1.5 and HUR interval = 2)

Fig. 5 summarizes the effect that workload and HUR update interval has on efficiency and to see why it is important to have a lower value of $k$ when the workload is high. In terms of the effect of the HUR update interval, we see that a larger interval does not have a noticeable impact on success and efficiency. Therefore, further network overhead can be saved by updating the HUR less frequently. Fig. 5 also shows that a clear but small difference in the effect of the update interval only appears for higher workloads. Similarly, Fig. 6 summarizes the effect of workload and HUR update interval on success ratio. Comparing Fig. 5 and Fig. 6, we can see that the success ratio is slightly increased for a higher update interval at the expense of reservation efficiency.

Finally, Fig. 7 shows the system effectiveness under different HUR tree heights. There is a definite tradeoff that arises from decreasing the tree height in the HUR when $k$ is below 0.7.

Fig. 5. Efficiency under (workload, HUR interval) variations (tree height = 10)

Fig. 6. Success ratio under (workload, HUR interval) variations (tree height = 10)

V. RELATED WORK

Real-time enterprise is an important business model that has received much interest from many IT companies. Microsoft has released Microsoft Dynamics as a line of integrated business management solutions that automate and streamline financial, customer relationship, and supply chain processes. IBM has proposed the complex event processing
(CEP) framework which is made up of WebSphere Business Events (the event processing engine), WebSphere Business Monitor, WebSphere Message Broker (event transformations and connectivity functions), and Generalized Publish and Subscribe Services (GPASS) [10]. The HP ZLE framework aims to provide application and data integration to create a seamless, enterprise-wide solution for real-time information and action. To our knowledge, however, none of the current real-time enterprise products offers the service reservation capability.

Mamat et. al. [11] discuss an advance reservation system for clusters, particularly to help manage I/O conflict among nodes in a cluster. They identify an advance factor for reservations that are multiples of task interarrival times, anywhere from immediate to a factor of ten. However, no special treatment of data structures would be required for such a short time range. Our research, however, requires efficient data structures for storing and managing reservations on a much longer time frame in advance.

The GARA system [12] is an advance reservation policy framework that is especially useful for reserving bandwidth for streaming applications. However, streaming media applications usually have more stable requirements than the individual service execution reservations required for dynamic RT-SOA. SNAP [13] also provides a powerful policy framework for flexible distributed reservations for a variety of resources. GRAM [14] is another policy framework that focuses on computational resources. RT-Llama offers a policy framework but also emphasizes the SOA middleware required for distributed resource reservations in order to have predictable service executions.

VI. Conclusion

In this paper, we propose the RT-Llama SOA framework that given a user’s request for an end-to-end deadline on a business process (BP) request, performs initial screening and feasibility checks based on cached utilization data, plans the full BP execution by efficiently reserving resources in advance on the hosts where each service is to execute. We have demonstrated such efficiency by showing that the screening process avoids unnecessary remote reservation operations. We also present our runtime scheduler design by using the VCPU mechanism so that each service class is temporally isolated from others without unpredictable interference. We have also discussed many features that we plan to implement in order to have a practical and complete RT-SOA framework.

References