SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems

Karl Czajkowski[1], Ian Foster[2,3], Carl Kesselman[1],
Volker Sander[4], and Steven Tuecke[2]

[1] Information Sciences Institute
University of Southern California, Marina del Rey, CA 90292 U.S.A.
{karlcz,carl}@isi.edu
[2] Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439 U.S.A.
{foster,tuecke}@mcs.anl.gov
[3] Department of Computer Science
The University of Chicago, Chicago, IL 60657 U.S.A.
[4] Zentralinstitut für Angewandte Mathematik
Forschungszentrum Jülich, 52425 Jülich, Germany

Abstract. A fundamental problem in distributed computing is to map activities such as computation or data transfer onto resources that meet requirements for performance, cost, security, or other quality of service metrics. The creation of such mappings requires negotiation among application and resources to discover, reserve, acquire, configure, and monitor resources. Current resource management approaches tend to specialize for specific resource classes, and address coordination across resources only in a limited fashion. We present a new approach that overcomes these difficulties. We define a resource management *model* that distinguishes three kinds of resource-independent service level agreements (SLAs), formalizing agreements to deliver capability, perform activities, and bind activities to capabilities, respectively. We also define a *Service Negotiation and Acquisition Protocol* (SNAP) that supports reliable management of remote SLAs. Finally, we explain how SNAP can be deployed within the context of the Globus Toolkit.

1 Introduction

A common requirement in distributed computing systems such as Grids [17, 20] is to negotiate access to, and manage, resources that exist within different administrative domains than the requester. Acquiring access to these remote resources is complicated by the competing needs of the *client* and the *resource owner*. The client needs to understand and affect resource behavior, often requiring assurance or guarantee on the level and type of service being provided by the resource. Conversely, the owner wants to maintain local control and discretion over how the resource can be used. Not only does the owner want to control usage policy,

D.G. Feitelson et al. (Eds.): JSSPP 2002, LNCS 2537, pp. 153–183, 2002.
© Springer-Verlag Berlin Heidelberg 2002

he often wants to restrict how much service information is exposed to clients. A common means for reconciling these two competing demands is to negotiate a *service-level agreement* (SLA), by which a resource provider "contracts" with a client to provide some measurable capability or to perform a task. An SLA allows clients to understand *what to expect from resources* without requiring detailed knowledge of competing workloads or resource owners' policies. This concept holds whether the managed resources are physical equipment, data, or logical services.

However, negotiation of SLAs for distributed Grid applications is complicated by the need to coordinate access to multiple resources simultaneously. For example, large distributed simulations [5] can require access to many large computational resources at one time. On-line experiments [41] require that computational resources be available when the experiment is being conducted, and processing pipelines such as data-transfer [22], data-analysis [26, 3] and visualization pipelines [9] require simultaneous access to a balanced resource set.

Given that each of the resources in question may be owned and operated by a different provider, establishing a single SLA across all of the desired resources is not possible. Our solution to this problem is to define a resource management model in which management functions are decomposed into different types of SLAs that can be composed incrementally, allowing for coordinated management across the desired resource set. Specifically, we propose three different types of SLAs:

– *Task service level agreements* (TSLAs) in which one negotiates for the performance of an activity or task. A TSLA is, for example, created by submitting a job description to a queuing system. The TSLA characterizes a task in terms of its service steps and resource requirements.
– *Resource service level agreements* (RSLAs) in which one negotiates for the right to consume a resource. An RSLA can be negotiated without specifying for what activity the resource will be used. For example, an advance reservation takes the form of an RSLA. The RSLA characterizes a resource in terms of its abstract service capabilities.
– *Binding service level agreements* (BSLAs) in which one negotiates for the application of a resource to a task. For example, an RSLA promising network bandwidth might be applied to a particular TCP socket, or an RSLA promising parallel computer nodes might be applied to a particular job task. The BSLA associates a task, defined either by its TSLA or some other unique identifier, with the RSLA and the resource capabilities that should be met by exploiting the RSLA.

As illustrated in Figure 1, the above SLAs define a resource management model in which one can submit tasks to be performed, get promises of capability, and lazily bind the two. By combining these agreements in different ways, we can represent a variety of resource management approaches including: batch submission, resource brokering, co-allocation and co-scheduling.

One concrete example of a lazily-established BSLA might be to increase the number of physical memory pages bound to a running process, based on

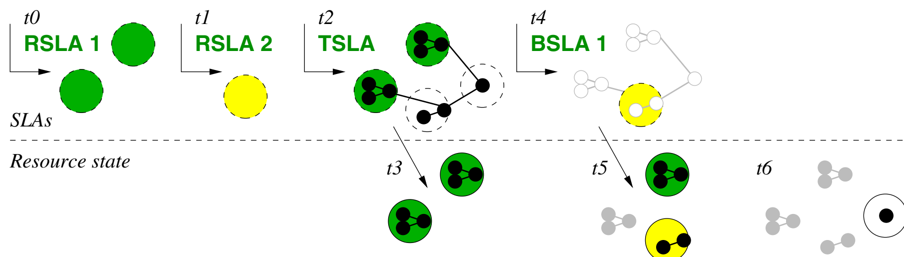

Fig. 1. Three types of SLA—RSLA, TSLA, and BSLA—allow a client to schedule resources as time progresses from t_0 to t_6. In this case, the client acquires two resource promises (RSLAs) for future times; a complex task is submitted as the sole TSLA, utilizing RSLA 1 to get initial portions of the job provisioned; later, the client applies RSLA 2 to accelerate provisioning of another component of the job; finally, the last piece of the job is provisioned by the manager without an explicit RSLA

observed data regarding the working-set size of the service. Another example is network QoS: a reservation regarding the path between two Internet host addresses may guarantee a client a minimum bandwidth flow as an RSLA. The client must bind TCP socket addresses to this reserved capability at runtime as a BSLA—the sockets are identifiable "tasks" most likely not managed with a TSLA. The complexity of real-world scenarios is addressed with combinations of such SLAs. The proposed SLA model is independent of the service being managed—the semantics of specific services are accommodated by the details of the agreement, and not in the types of agreements negotiated. Because of its general applicability, we refer to the protocols used to negotiate these SLAs as the *Service Negotiation and Acquisition Protocol* (SNAP).

The service management approach proposed here extends techniques first developed within the Globus Toolkit's GRAM service [8] and then extended in the experimental GARA system [21, 22, 36]. An implementation of this architecture and protocol can leverage a variety of existing infrastructure, including the Globus Toolkit's Grid Security Infrastructure [19] and Monitoring and Discovery Service [10]. We expect the SNAP protocol to be easily implemented within the Open Grid Services Architecture (OGSA) [18, 39], which provides request transport, security, discovery, and monitoring.

The remainder of this paper has the following structure: in Section 2 we present several motivating scenarios to apply SLA models to Grid RM problems; in Section 3 we present the SNAP protocol messages and state model, which embed a resource and task language characterized in Section 4. In Section 5, we briefly formalize the relationship between the various SLA and resource languages in terms of their satisfaction or solution spaces. Finally, in Sections 6 and 7, we describe how SNAP can be implemented in the context of Globus services and relate it to other QoS and RM work.

2 Motivating Scenarios

The SNAP SLA model is designed to address a broad range of applications through the aggregation of simple SLAs. In this section we examine two common scenarios: a Grid with "community schedulers" mediating access to shared resources on behalf of different client groups, and a file-transfer scenario where QoS guarantees are exploited to perform data staging under deadline conditions.

2.1 Community Scheduler Scenario

A community scheduler (sometime referred to as a resource broker) is an entity that acts as an intermediary between the community and its resources: activities are submitted to the community scheduler rather than to the end resource, and the activities are scheduled onto community resources in such as way as to optimize the community's use of its resource set.

As depicted in Figure 2, a Grid environment may contain many resources (R1–R6), all presenting an RSLA interface as well as a TSLA interface. Optimizing the use of resources across the community served by the scheduler is only possible if the scheduler has some control over the resources used by the community. Hence the scheduler negotiates capacity guarantees via RSLAs with a pool of underlying resources, and exploits those capabilities via TSLAs and BSLAs. This set of agreements abstracts away the impact of other community schedulers as well as any "non-Grid" local workloads, assuming the resource managers enforce SLA guarantees at the resources.

Community scheduler services (S1 and S2 in Figure 2) present a TSLA interface to users. Thus a community member can submit a task to the scheduler by negotiating a TSLA, and the scheduler in turn hands this off to a resource by

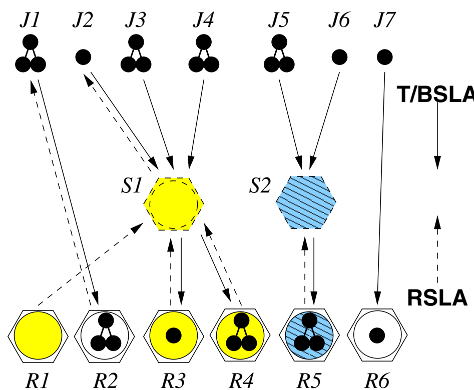

Fig. 2. Community scheduler scenario. Multiple users (J1–J7) gain access to shared resources (R1–R6). Community schedulers (S1–S2) mediate access to the resources by making TSLAs with the users and in turn making RSLAs and TSLAs with the individual resources

binding this TSLA against one of the existing RSLAs. The scheduler may also offer an RSLA interface. This would allow applications to co-scheduler activities across communities, or combine community scheduled resources with additional non-community resources.

The various SLAs offered by the community scheduler and underlying resources result in a very flexible resource management environment. Users in this environment interact with community and resource-level schedulers as appropriate for their goals and privileges. A privileged client with a batch job such as J7 in Figure 2 may not need RSLAs, nor the help of a community scheduler, because the goals are expressed directly in the TSLA with resource R6. The interactive job J1 needs an RSLA to better control its performance. Jobs J2 to J6 are submitted to community schedulers S1 and S2 which might utilize special privileges or domain-specific knowledge to efficiently implement their community jobs. Note that not all users require RSLAs from the community scheduler, but S1 does act as an RSLA "reseller" between J2 and resource R3. Scheduler S1 also maintains a speculative RSLA with R1 to more rapidly serve future high-priority job requests.

2.2 File Transfer Scenarios

In these scenarios, we consider that the activity requested by the user is to transfer a file from one storage system to another. Generalizing the community scheduler example, we augment the behavior of the scheduler to understand that a transfer requires storage space on the destination resource, and network and endpoint I/O bandwidth during the transfer. The key to providing this service is the ability of the scheduler to manage multiple resource types and perform co-scheduling of these resources.

File Transfer Service As depicted in Figure 3, the file transfer scheduler S1 presents a TSLA interface, and a network resource manager R2 presents an RSLA interface. A user submits a transfer job such as J1 to the scheduler with a deadline. The scheduler obtains a storage reservation on the destination resource R3 to be sure that there will be enough space for the data before attempting the transfer. Once space is allocated, the scheduler obtains bandwidth reservations from the network and the storage devices, giving the scheduler confidence that the transfer can be completed within the user-specified deadline. Finally, the scheduler submits transfer endpoint jobs J2 and J3 to implement the transfer J1 using the space and bandwidth promises.

Job Staging with Transfer Service SLAs can be linked together to address more complex resource co-allocation situations. We illustrate this considering a job that consists of a sequence of three activities: data is transferred from a storage system to an intermediate location, some computation is performed using the data, and the result is transferred to a final destination. The computation is performed on resources allocated to a community of users. However, for

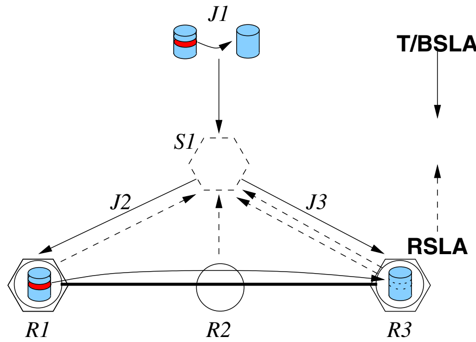

Fig. 3. File transfer scenario. File transfer scheduler coordinates disk and network reservations before co-scheduling transfer endpoint jobs to perform transfer jobs for clients

security reasons, the computation is not performed using a group account, but rather, a temporary account is dynamically created for the computation (In [32], we describe a community authorization service which can be used to authorize activities on behalf of a user community).

In Figure 4, TSLA1 represents a temporary user account, such as might be established by a resource for a client who is authorized through a Community Authorization Service. All job interactions by that client on the resource become linked to this long-lived SLA—in order for the account to be reclaimed safely, all dependent SLAs must be destroyed. The figure illustrates how the individual SLAs associated with the resources and tasks can be combined to address the end-to-end resource and task management requirements of the entire job. Of interest in this example are:

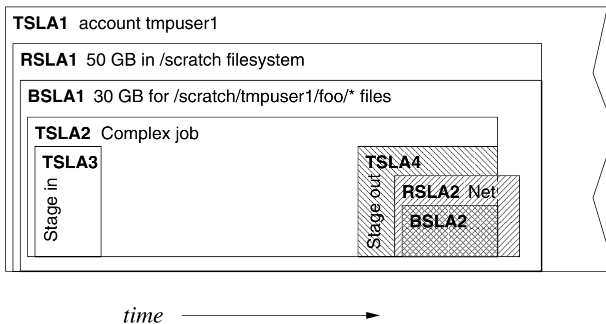

Fig. 4. Dependent SLAs for file transfers associated with input and output of a job with a large temporary data space. BSLA2 is dependent on TSLA4 and RSLA2, and has a lifetime bound by those two

TSLA1 is the above-mentioned temporary user account.
RSLA1 promises the client 50 GB of storage in a particular file-system on the resource.
BSLA1 binds part of the promised storage space to a particular set of files within the file-system.
TSLA2 runs a complex job which will spawn constituent parts for staging of input and output data.
TSLA3 is the first file transfer task, to stage the input to the job site without requiring any additional QoS guarantees in this case.
TSLA4 is the second file transfer task, to stage the large output from the job site, under a deadline, before the local file-system space is lost.
RSLA2 and BSLA2 are used by the file transfer service to achieve the additional bandwidth required to complete the (large) transfer before the deadline.

The job scheduled by TSLA2 might have built-in logic to establish the staging jobs TSLA3 and TSLA4, or this logic might be part of the provider performing task TSLA2 on behalf of the client. In the figure, the nesting of SLA "boxes" is meant to illustrate how lifetime of these management abstractions can be linked in practice. Such linkage can be forced by a dependency between the subjects of the SLAs, e.g. BSLA2 is meaningless beyond the lifetime of TSLA4 and RSLA2, or optionally added as a management convenience, e.g. triggering recursive destruction of all SLAs from the root to hasten reclamation of application-grouped resources.

2.3 Resource Virtualization

In the preceding scenarios, the Community Scheduler can be viewed as virtualizing a set of resources from other managers for the benefit of its community of users. This type of resource virtualization is important as it helps implement the trust relationships that are exploited in Grid applications. The user community trusts their scheduler to form agreements providing resources (whether basic hardware capabilities or complex service tasks), and the scheduler has its own trust model for determining what resources are acceptable targets for the community workload.

Another type of virtualization in dynamic service environments like the Open Grid Service Architecture (OGSA) is captured in the *factory* service model [18]. A SNAP manager in such an environment *produces* SLAs, providing a long-lived contact point to initiate and manage the agreements. The SLA factory exposes the agreements as set of short-lived, stateful services which can be manipulated to control one SLA. Resource virtualization is particularly interesting when a TSLA schedules a job which can itself provide Grid services. This process is described for "active storage" systems in [26] and [9], where data extraction jobs convert a compute cluster with parallel storage into an application-specialized data server. The submission of a TSLA running such jobs can be thought of as the dynamic deployment of new services "on demand," a critical property for a permanent, but adaptive, global Grid [20].

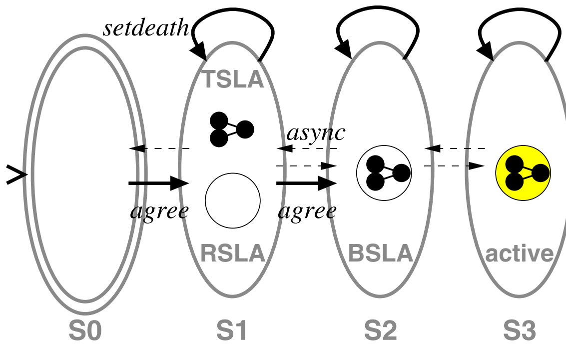

Fig. 5. Agreement state transitions. State of SLAs is affected by client requests (solid arrows) and other internal behaviors in the manager (dashed arrows)

3 The SNAP Agreement Protocol

The core of the SNAP architecture is a client-service interaction used to negotiate SLAs. The protocol applies equivalently when talking to authoritative, localized resource owners or to intervening brokers.

We describe each operation in terms of unidirectional messages sent from client to service or service to client. All of these operations follow a client-server remote procedure-call (RPC) pattern, so we assume the underlying transport will provide correlation of the initiating and responding messages. One way of interpreting the following descriptions is that the client to service message corresponds to the RPC, and the return messages represent the possible result values of the call. This interpretation is consistent with how such a protocol would be deployed in a Web Services environment, using WSDL to model the RPC messages [7, 1].

3.1 Agreement State Transitions

Due the the dependence of BSLAs on RSLAs (and possibly on TSLAs), there are four states through which SNAP progresses, as depicted in Figure 5:

S0: SLAs either have not been created, or have been resolved by expiration or cancellation.
S1: Some TSLAs and RSLAs have been agreed upon, but may not be bound to one another.
S2: The TSLA is matched with the RSLA, and this grouping represents a BSLA to resolve the task.
S3: Resources are being utilized for the task and can still be controlled or changed.

As indicated in Figure 5 with solid arrows, client establishment of SLAs enters the state S1, and can also lead to state S2 by establishing BSLAs. It is possible for

the manager to unilaterally create a BSLA representing its schedule for satisfying a TSLA, and only the manager can move from a BSLA into a run-state S3 where resources are actively supporting a task. Either client termination requests, task completion, or faults may lead back to a prior state, including termination or failure of SLAs in state S0.

3.2 Agreement Meta-language

The SNAP protocol maintains a set of manager-side SLAs using client-initiated messages. All SLAs contain an SLA identifier I, the client c with whom the SLA is made, and an expiration time t_{dead}, as well as a specific TSLA, RSLA, or BSLA description d:

$$\langle I, c, t_{\text{dead}}, d \rangle.$$

Each SLA type defines its own descriptive content, e.g. resource requirements or task description. In this section we assume an extensible language $\mathbf{J}$ for describing tasks (jobs), with a subset language $\mathbf{R} \subseteq \mathbf{J}$ capable of expressing resource requirements in $\mathbf{J}$ as well as apart from any specific task description. The necessary features of such a language are explored later in Section 4.

We also assume a relation $a' \sqsubseteq a$, or a' *models* a, which means that a' describes the same terms of agreement as a but might possible add additional terms or further restrict a constraint expressed in a. In other words, any time SLA a' conditions are met, so are a conditions. This concept is examined more closely in Section 5.

RSLA Content An RSLA contains the (potentially complex) resource capability description r expressed in the $\mathbf{R}$ subset of the $\mathbf{J}$ language. Therefore, a complete RSLA in a manager has the form:

$$\langle I, c, t_{\text{dead}}, \langle r \rangle_{\text{R}} \rangle.$$

TSLA Content A TSLA contains the (potentially complex) job description j expressed in the $\mathbf{J}$ language. Therefore, a complete TSLA in a manager has the form:

$$\langle I, c, t_{\text{dead}}, \langle j \rangle_{\text{T}} \rangle.$$

The description j also includes a resource capability description $r = j \downarrow_{\mathbf{R}}$ which expresses what capability r is to be applied to the task, and using what RSLA(s). If the named RSLAs are not sufficient to satisfy r, the TSLA implies the creation of one or more RSLAs to satisfy j.

BSLA Content A BSLA contains the description j of an existing task in the language $\mathbf{J}$. The description j may reference a TSLA for the task, or some other unique description in the case of tasks not initiated by a TSLA. Therefore, a complete stand-alone RSLA in a manager has the form:

$$\langle I, c, t_{\text{dead}}, \langle j \rangle_{\text{B}} \rangle.$$

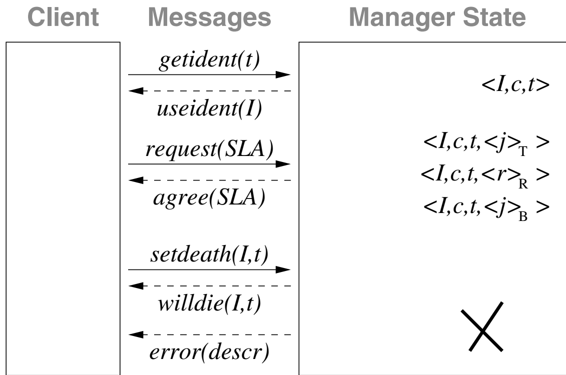

Fig. 6. RM protocol messages. The protocol messages establish and maintain SLAs in the manager

As for TSLAs, the BSLA description j may reference existing RSLAs and if they do not satisfy the requirements in j, the BSLA implies the creation of one or more RSLAs to satisfy j.

3.3 Operations

Allocate Identifier Operation There are multiple approaches to obtaining unique identifiers suitable for naming agreements. To avoid describing a security infrastructure-dependent approach, we suggest a special light-weight agreement to allocate identifiers from a manager. This operation is analogous to opening a timed transaction in a database system. The client sends:

$$\textbf{getident}(t_{\text{dead}}),$$

asking the manager to allocate a new identifier that will be valid until time t_{dead}. On success, the manager will respond:

$$\textbf{useident}(I, t_{\text{dead}}),$$

and the client can then attempt to create reliable RM agreements using this identifier as long as the identifier is valid. A common alternative approach would fold the identifier allocation into an initial SLA request, requiring a follow-up acknowledgment or *commit* message from the client to complete the agreement. With the above separation of identifier allocation, we avoid confusing this reliable messaging problem with a different multi-phase negotiation process inherent in distributed co-reservation (where the concept of "commitment" is more generally applicable).

Agreement Operation A client negotiates an SLA using a valid identifier obtained using **getident**(...). The client issues a single message with arguments expressed in the agreement language from Section 3.2:

$$\textbf{request}(I, c, t_{\text{dead}}, a).$$

The SLA description a captures all of the requirements of the client. On success, the manager will respond with a message of the form:

$$\textbf{agree}(I, c, t_{\text{dead}}, a'),$$

where $a' \sqsubseteq a$ as described in Sections 3.2 and 5. In other words, the manager agrees to the SLA description a', and this SLA will terminate at t_{dead} unless the client performs a **setdeath**(I, t) operation to change its scheduled lifetime.

A client is free to re-issue requests, and a manager is required to treat duplicate requests received after a successful agreement as being equivalent to a request for acknowledgment on the existing agreement. This idempotence is enabled by the unique identifier of each agreement.

Set Termination Operation We believe that *idempotence* (i.e. an at-most-once semantics) combined with *expiration* is well-suited to achieving fault-tolerant agreement. We define our operations as atomic and idempotent interactions that create SLAs in the manager. Each SLA has a termination time, after which a well-defined reclamation effect occurs. This termination effect can be exploited at runtime to implement a spectrum of negotiation strategies: a stream of short-term expiration updates could implement a heart-beat monitoring system [37] to force reclamation in the absence of positive signals, while a long-term expiration date guarantees SLAs will persist long enough to survive transient outages.

With this operation, a client can set a new termination time for the identifier (and any agreement named as such). The client changes the lifetime by sending a message of the form:

$$\textbf{setdeath}(I, t_{\text{dead}}),$$

where t_{dead} is the new wall-clock termination time for the existing SLA labeled by I. On success the manager will respond with the new termination time:

$$\textbf{willdie}(I, t_{\text{dead}}),$$

and the client may reissue the **setdeath**(...) message if some failure blocks the initial response. Agreements can be abandoned with a simple request of **setdeath**$(I, 0)$ which forces expiration of the agreement.

The lifetime represented by t_{dead} is the lifetime of the agreement named by I. If the agreement makes promises about times in the future beyond its current lifetime, those *promises expire with the SLA*. Thus, it is a client's responsibility to extend or renew an SLA for the full duration required.

3.4 Change

Finally, we support the common idiom of atomic *change* by allowing a client to resend the request on the same SLA identifier, but with modified requirement content. The service will respond as for an initial request, or with an error if the given change is not possible from the existing SLA state. When the response indicates a successful SLA, the client knows that any preceding agreement named by I has been replaced by the new one depicted in the response. When the response indicates failure, the client knows that the state is unchanged from before the request.

In essence, the service compares the incoming SLA request with its internal policy state to determine whether to treat it as a *create*, *change*, or *lookup*. The purpose of change semantics is to preserve state in the underlying resource behavior where that is useful, e.g. it is often possible to preserve an I/O channel or compute task when QoS levels are adjusted. Whether such a change is possible may depend both on the resource type, implementation, and local policy. If the change is refused, the client will have to initiate a new request and deal with the loss of state through other means such as task check-pointing. An alternative to implicit change would be an explicit change mechanism to perform structural *editing* of the existing SLA content, but we do not define concrete syntax for the **R** and **J** languages as would be needed to formalize such editing.

Change is also useful to adjust the degree of commitment in an agreement. An expected use is to monotonically increase the level of commitment in a promise (or cancel it) as a client converges on an application schedule involving multiple resource managers. This use essentially implements a timed, multi-phase commit protocol across the managers which may be in different administrative domains. However, there is no architectural requirement for this monotonic increase—a client may also want to decrease the level of commitment if they lose confidence in their application plan and want to relax agreements with the manager.

4 Resource and Task Meta-language

The resource and scheduling language **J** assumed in Section 3 plays an important role in our architecture. Clients in general must request resources by property, e.g. by capability, quality, or configuration. Similarly, clients must understand their assignments by property so that they can have any expectation of delivery in an environment where other clients' assignments and activities may be hidden from view.

In this section we examine some of the structures we believe are required in this language, without attempting to specify a concrete syntax. As a general note, we believe that resource description must be dynamically extensible, and the correct mechanism for extension is heavily dependent on the technology chosen to implement SNAP. Sets of clients and resources must be able to define new resource syntax to capture novel devices and services, so the language should support these extensions in a structured way. However, a complex

new concept may sometimes be captured by composing existing primitives, and hopefully large communities will be able to standardize a relatively small set of such composeable primitives.

4.1 Resource Metrics

Many resources have parameterized attributes, i.e. a metric describing a particular property of the resource such as *bandwidth*, *latency*, or *space*. Descriptions may scope these metrics to a window of time $[t_0, t_1]$ in which the client desires access to a resource with the given qualities. We use a generic scalar metric and suggest below how they can be composed to model conventional resources.

A scalar metric can exactly specify resource capacity. Often requirements are partially constraining, i.e. they identify ranges of capacity. We extend scalar metrics as unary inequalities to use the scalar metrics as a *limit*. The limit syntax can also be applied to time values, e.g. to specify a start time of "$\leq t$" for a provisioning interval that starts "on or before" the exact time t.

Time metrics t expressed in wall-clock time, e.g. "Wed Apr 24 20:52:36 UTC 2002."

Scalar metrics $x\,u$ expressed in x real-valued units u, e.g. 512 Mbytes, or 10×10^{-3} s/seek.

Max limit $< m$ and $\leq m$ specify an exclusive or inclusive upper limit on the given metric m, respectively.

Min limit $> m$ and $\geq m$ specify an exclusive or inclusive lower limit on the given metric m, respectively.

These primitives are "leaf" constructs in a structural resource description. They define a syntax, but some of their meaning is defined by the context in which they appear.

4.2 Resource Composites

The resource description language is compositional. Realistic resources can be modeled as composites of simpler resource primitives. Assuming a representation of resources r_1, r_2 etc. we can aggregate them using various typed constructs.

Set $[r_1, r_2, \ldots]$ combining arbitrary resources that are all required.

Typed Set $[r_1, r_2, \ldots]_{type}$ combining type-specific resources. Groups are marked with a *type* to convey the meaning of the collection of resources, e.g. $[x_1$ bytes, x_2 bytes/s$]_{disk}$ might collect space and bandwidth metrics for a "file-system" resource.

Array $n \times r$ is an abbreviation for the group of n identical resource instances $[\,r, r, \ldots, r]$, e.g. for convenient expression of symmetric parallelism.

The purpose of typed groups is to provide meaning to the metric values inside—in practice the meaning would be denoted only in an external specification of

the type, and the computer system interrogating instances of R will be implemented to recognize and process the typed composite. For example, the $[x_1\,\text{bytes}, x_2\,\text{bytes/s}]_{\text{disk}}$ composite tells us that we are constraining the speed and size of a secondary storage device with the otherwise ambiguous metrics for space and bandwidth.

Resources are required over periods of time, i.e. from a start time t_0 to an end time t_1, and we denote this as $r^{[t_0,t_1]}$. A complex time-varying description can be composed of a sequence of descriptions with consecutive time intervals:

$$r = \left[[r_1]^{[t_0,t_1]}, [r_2]^{[t_1,t_2]}, \ldots, [r_n]^{[t_{n-1},t_n]} \right]^{[t_0,t_n]}.$$

Each subgroup within a composite must have a lifetime wholly included within the lifetime of the parent group.

4.3 Resource Alternatives

We define disjunctive *alternatives* to complement the conjunctive composites from section 4.2.

Alternative $\vee(r_1, r_2, \ldots)$ differs from a resource set in that only one element r_i must be satisfied.

As indicated in the descriptions above, limit modifiers are only applicable to scalar metrics, while the alternative concept applies to all resource description elements. Alternatives can be used to express alternate solution spaces for the application requirements within distinct planning regimes, or to phrase similar requirements using basic and specialized metrics in the event that a client could benefit from unconventional extensions to **J** that may or may not be recognized by a given manager.

4.4 Resource Configuration

The final feature present in our description language is the ability to intermingle control or *configuration* directives within the resource statement. In an open environment, this intermingling is merely a notational convenience to avoid presenting two isomorphic statements—one modeling the requirements of the structured resource and one providing control data to the resource manager for the structured resource. Task configuration details are what are added to the language **R** to define the activity language **J** .

Configure $a := v$ specifies an arbitrary configuration attribute a should have value v.

In an environment with limited trust and strict usage restrictions, some resources may be unavailable for certain configurations due to owner policy. We therefore suggest treating them as primitive metrics when considering the meaning of the description for resource selection, while also considering them as control data when considering the meaning of the description as an activity configuration.

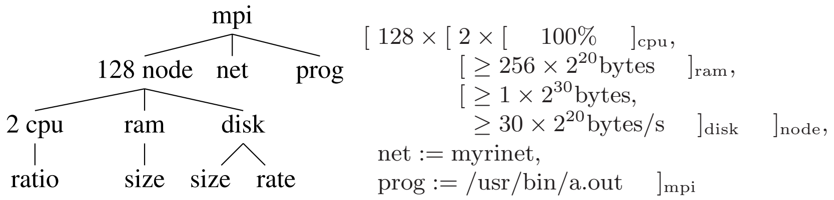

Fig. 7. Hypothetical resource description. A parallel computer with 128 dedicated dual-processor nodes, each providing at least 256 MB of memory and 1 GB disk with disk performance of 30 MB/s, connected by Myrinet-enabled MPI. A parse tree is provided to help illustrate the nested expression

4.5 RSLA Binding

To support the referencing of RSLAs, we require a way to associate an existing RSLA with a sub-requirement in $\mathbf{J}$:

RSLA Binding $[r, I_\mathrm{B}]_\mathrm{bind}$ specifies requirement r but also says it should be satisfied using the RSLA identified by I_B.

This construct supports the explicit resource planning described in Section 3.2.

5 SLA Constraint-Satisfaction Model

In a fully-developed SLA environment, one can imagine agreements including auditing commitments, negotiated payments or exchange of service, and remediation steps in case of agreement violation. However, in this paper we focus on a weaker form of agreement where clients more or less trust resource providers to act in good faith, and cost models for service are not explicitly addressed nor proscribed. Nonetheless, the entire purpose of our protocol hinges on an understanding of *satisfaction* of SNAP SLAs. The satisfaction of an SLA requires a non-empty "solution set" of possible resource and task schedules which deliver the capabilities and perform the directives encoded in the $\mathbf{J}$ language elements within the SLA. A self-contradictory or *unsatisfiable* SLA has an empty solution set. We denote the ideal solution set with *solution operators* $S_\mathbf{R}(r)$ and $S_\mathbf{J}(j)$ which apply to descriptions in $\mathbf{R}$ or $\mathbf{J}$.

While the language $\mathbf{R}$ is assumed to be a syntactic subset of $\mathbf{J}$, the set of solution sets $\{S_\mathbf{R}(r) \mid r \in \mathbf{R}\}$ is a *superset* of the set of solution sets $\{S_\mathbf{J}(j) \mid j \in \mathbf{J}\}$, and given a projection of requirements $j \downarrow_\mathbf{R} \in \mathbf{R}$, the solution set $S_\mathbf{R}(j \downarrow_\mathbf{R})$ is a superset of $S_\mathbf{J}(j)$. This inversion occurs because the additional syntactic constructs in $\mathbf{J}$ are used to express additional task constraints beyond the resource capabilities expressible in $\mathbf{R}$. We would like a relation between descriptions to capture this relationship between solution sets for the descriptions. We say that a *refined* description j' *models* j, or $j' \sqsubseteq j$, if and only if $S_\mathbf{J}(j') \subseteq S_\mathbf{J}(j)$. This concept of refinement is used to define the relationship between requested and agreed-upon SLAs in the SLA negotiation of Section 3.3.

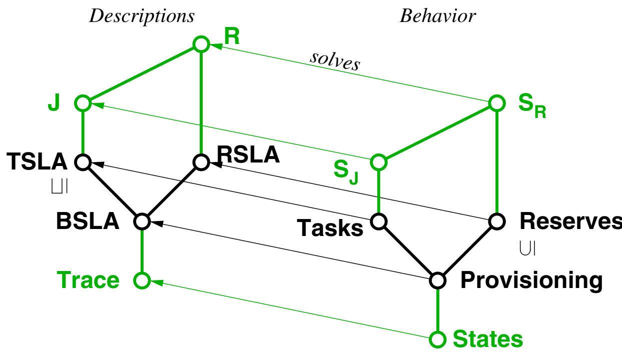

Fig. 8. Constraint domain. Lower items in the figure conservatively approximate higher items. The solution spaces on the right are ordered as subsets, e.g. Provisioning⊆Reserves because provisioning constrains a resource promise to a particular task. Solution ordering maps to the "model" relation for constraints, e.g. BSLA⊑RSLA on the left

Just as **J** is more expressive than **R** , BSLAs are more expressive than TSLAs or RSLAs. The TSLA says that a manager will "run job j according to its self-expressed performance goals and provisioning requirements." The RSLA says that a manager will "provide resource capability r when asked by the client." A corresponding BSLA encompasses both of these and says the manager will "apply resource r to help satisfy requirements while performing job j." Therefore we extend our use of the "models" relation to SLAs. This set-ordered structure in the SNAP concept domain is illustrated in Figure 8.

6 Implementing SNAP

The RM protocol architecture described in this article is general and follows a minimalist design principle in that the protocol captures only the behavior that is essential to the process of negotiation. We envision that SNAP would not be implemented as a stand alone protocol, but in practice would be layered on top of more primitive protocols and services providing functions such as communication, authentication, naming, discovery, etc. For example, the Open Grid Services Architecture [18] defines basic mechanisms for creating, naming, and controlling the lifetime of services. In the following, we explore how SNAP could be implemented on top of the OGSA service model.

6.1 Authentication and Authorization

Because Grid resources are both scarce and shared, a system of rules for resource use, or *policy*, is often associated with a resource to regulate its use [40]. We assume a wide-area security environment such as GSI [19] will be integrated

with the OGSA to provide mutually-authenticated identity information to SNAP managers such that they may securely implement policy decisions. Both upward information flow and downward agreement policy flow in a complex service environment, such as depicted in Figure 9, are likely subject to policy evaluation that distinguishes between individual clients and/or requests.

6.2 Resource Heterogeneity

The SNAP protocol agreements can be mapped onto a range of existing local resource managers, to deploy its beneficial capabilities without requiring wholesale replacement of existing infrastructure. Results from GRAM testbeds have shown the feasibility of mapping TSLAs onto a range of local job schedulers, as well as simple time-sharing computers [16, 6, 35]. The GARA prototype has shown how RSLAs and BSLAs can be mapped down to contemporary network QoS systems [21, 22, 36]. Following this model, SNAP manager services represent *adaptation* points between the SNAP protocol domain and local RM mechanisms.

6.3 Monitoring

A fundamental function for RM systems is the ability to monitor the health and status of individual services and requests. Existing Grid RM services such as GRAM and GARA include native protocol features to signal asynchronous state changes from a service to a client. In addition to these native features, some RM state information is available from a more generalized information service, e.g. GRAM job listings are published via the MDS in the Globus Toolkit [8, 21, 10].

We expect the OGSA to integrate asynchronous subscription/notification features. Therefore, we have omitted this function from the RM architecture presented here. An RM service implementation is expected to leverage this common infrastructure for its monitoring data path. We believe the agreement model presented in Sections 1, 3.2 and 3.1 suggest the proper structure for exposing RM service state to information clients, propagating through the upward arrows in Figure 9. Information *index services* can cache and propagate this information because life-cycle of the agreement state records is well defined in the RM protocol semantics, and the nested request language allows detailed description of agreement properties.

6.4 Resource and Service Discovery

SNAP relies on the ability for clients to discover RM services. We expect SNAP services to be discovered via a combination of general discovery and registry services such as the index capabilities of MDS-2 and OGSA, client configuration via service registries such as UDDI, and static knowledge about the community (Virtual Organization) under which the client is operating. The discovery

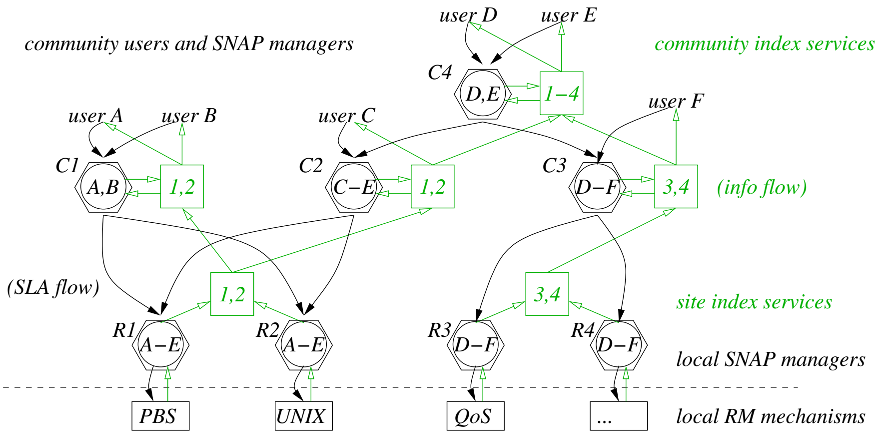

Fig. 9. An integrated SNAP system. Discovery services provide indexed views of resources, while SNAP managers provide distributed and aggregated resource brokering abstractions to users

information flow is exactly as for monitoring in Figure 9, with information propagating from resources upward through community indexes and into clients. In fact, discovery is one of the purposes for a general monitoring infrastructure.

Due to the potential for virtualized resources described in Section 2.3, we consider "available resources" to be a secondary capability of "available services." While service environments provide methods to map from abstract service names to protocol-level service addresses, it is also critical that services be discoverable in terms of their capabilities. The primary capability of a SNAP manager is the set of agreements it *offers*, i.e. that it is willing to establish with clients.

6.5 Multi-phase Negotiation

There are dynamic capabilities that also restrict the agreement space, including resource load and RM policy. Some load information may be published to help guide clients with their resource selection. However, proprietary policy including priorities and hidden SLAs may effect availability to specific classes of client.

The agreement negotiation itself is a discovery process by which the client determines the willingness of the manager to serve the client. By formulating future agreements with weak commitment and changing them to stronger agreements, a client is able to perform a multi-phase commit process to discover more information in an unstructured environment. Resource virtualization helps discovery by aggregating policy knowledge into a private discovery service—a community scheduler can form RSLAs with application service providers and then expose this virtual resource pool through community-specific agreement offers.

6.6 Standard Modeling Language

In Section 4 we present the abstract requirements of an expressive resource language **J** . These requirements include unambiguous encoding of provisioning metrics, job configuration, and composites. We also identify above the propagation of resource and agreement state through monitoring and discovery data paths as important applications of the resource language. For integration with the OGSA, we envision this language **J** being defined by an XML-Schema [14] permitting extension with new composite element types and leaf metric types. The name-space features of XML-Schema permit unambiguous extension of the language with new globally-defined types.

This language serves the same purpose as RSL in GRAM/GARA [8, 11, 21, 22] or Class Ads in Condor [34, 27]. With SNAP, we are proposing a more extensible model for novel resource composites than RSL and a more rigorously typed extension model than Class Ads, two features which we believe are necessary for large-scale, inter-operable deployments.

6.7 Agreement Delegation

In the preceding protocol description, mechanisms are proposed to negotiate agreement regarding activity implementation or. These agreements capture a *delegation* of resource or responsibility between the negotiating parties. However, it is important to note that the delegation concept goes beyond these explicit agreements. There are analogous implicit delegations that also occur during typical RM scenarios.

The TSLA delegates specific task-completion responsibilities to the scheduler that are “held” by the user. The scheduler becomes responsible for reliably planning and enacting the requested activity, tracking the status of the request, and perhaps notifying the user of progress or terminal conditions. The RSLA delegates specific resource capacity to the user that are held by the manager. Depending on the implementation of the manager, this delegation might be mapped down into one or more hidden operational policy statements that enforce the conditions necessary to deliver on the guarantee. For example, a CPU reservation might prevent further reservations from being made or an internal scheduling priority might be adjusted to “steal” resources from a best-effort pool when necessary.

Transfers of rights and responsibilities are transitive in nature, in that an entity can only delegate that which is delegated to the entity. It is possible to form RSLAs out of order, but in order to exploit an RSLA, the dependent RSLAs must be valid. Such transitive delegation is limited by availability as well as trust between RM entities. A manager which over-commits resources will not be able to make good on its promises if too many clients attempt to use the RSLAs at the same time. Viewing RSLAs and TSLAs as delegation simplifies the modeling of heavy-weight brokers or service providers, but it also requires a trust/policy evaluation in each delegation step. A manager may restrict its delegations to only permit certain use of the resource by a client—this client may attempt to

broker the resource to other clients, but those clients will be blocked when they try to access the resource and the manager cannot validate the delegation chain.

6.8 Many Planners

Collective resource scenarios are the key motivation for Grid RM. In our architecture, the local resource managers do not solve these collective problems. The user, or an agent of the user, must obtain capacity delegations from each of the relevant resource managers in a resource chain. There are a variety of brokering techniques which may help in this situation, and we believe the appropriate technique must be chosen by the user or community. The underlying Grid RM architecture must remain open enough to support multiple concurrent brokering strategies across resources that might be shared by multiple user communities.

7 Other Related Work

Numerous researchers have investigated approaches to QoS delivery [23] and resource reservation for networks [12, 15, 42], CPUs [25], and other resources.

Proposals for advance reservations typically employ cooperating servers that coordinate advance reservations along an end-to-end path [42, 15, 12, 24]. Techniques have been proposed for representing advance reservations, for balancing immediate and advance reservations [15], for advance reservation of predictive flows [12]. However, this work has not addressed the co-reservation of resources of different types.

The Condor high-throughput scheduler can manage network resources for its jobs. However, it does not interact with underlying network managers to provide service guarantees [2] so this solution is inadequate for decentralized environments where network admission-control cannot be simulated in this way by the job scheduler.

The concept of a bandwidth broker is due to Jacobson. The Internet 2 Qbone initiative and the related Bandwidth Broker Working Group are developing testbeds and requirements specifications and design approaches for bandwidth brokering approaches intended to scale to the Internet [38]. However, advance reservations do not form part of their design. Other groups have investigated the use of differentiated services (e.g., [43]) but not for multiple flow types. The co-reservation of multiple resource types has been investigated in the multimedia community: see, for example, [28, 31, 30]. However, these techniques are specialized to specific resource types.

The Common Open Policy Service (COPS) protocol [4] is a simple protocol for the exchange of policy information between a Policy Decision Point (PDP) and its communication peer, called Policy Enforcement Point (PEP). Communication between PEP and PDP is done by using a persistent TCP connection in the form of a stateful request/decision exchange. COPS offers a flexible and extensible mechanism for the exchange of policy information by the use of the client-type object in its messages. There are currently two classes of COPS client:

Outsourcing provides an asynchronous model for the propagation of policy decision requests. Messages are initiated by the PEP which is actively requesting decisions from its PDP.

Provisioning in COPS follows a synchronous model in which the policy propagation is initiated by the PDP.

Both COPS models map easily to SNAP with the SNAP manager as a PDP and the resource implementation as a PEP. A SNAP client can also be considered a PDP which provisions policy (SLAs) to a SNAP manager which is then the PEP. There is no analogue to COPS outsourcing when considering the relationship between SNAP clients and managers.

7.1 GRAM

The Globus Resource Allocation Manager (GRAM) provides job submission on distributed compute resources. It defines APIs and protocols that allow clients to securely instantiate job running agreements with remote schedulers [8]. In [11], we presented a light-weight, opportunistic broker called DUROC that enabled simultaneous co-allocation of distributed resources by layering on top of the GRAM API. This broker was used extensively to execute large-scale parallel simulations, illustrating the challenge of coordinating computers from different domains and requiring out-of-band resource provisioning agreements for the runs [5, 6]. In exploration of end-to-end resource challenges, this broker was more recently used to acquire clustered storage nodes for real-time access to large scientific datasets for exploratory visualization [9].

7.2 GARA

The General-purpose Architecture for Reservation and Allocation (GARA) provides advance reservations and end-to-end management for quality of service on different types of resources, including networks, CPUs, and disks [21, 22]. It defines APIs that allows users and applications to manipulate reservations of different resources in uniform ways. For networking resources, GARA implements a specific network resource manager which can be viewed as a bandwidth broker.

In [36], we presented a bandwidth broker architecture and protocol that addresses the problem of diverse trust relationships and usage policies that can apply in multi-domain network reservations. In this architecture, individual BBs communicate via bilaterally authenticated channels between peered domains. Our protocol provides the secure transport of requests from source domain to destination domain, with each bandwidth broker on the path being able to enforce local policies and modify the request with additional constraints. The lack of a transitive trust relation between source- and end-domain is addressed by a delegation model where each bandwidth broker on the path being able to identify all upstream partners by accessing the credentials of the full delegation chain.

8 Conclusions

We have presented a new model and protocol for managing the process of negotiating access to, and use of, resources in a distributed system. In contrast to other architectures that focus on managing particular types of resources (e.g., CPUs or networks), our Service Negotiation and Acquisition Protocol (SNAP) defines a general framework within which reservation, acquisition, task submission, and binding of tasks to resources can be expressed for any resource in a uniform fashion.

We have not yet validated the SNAP model and design in an implementation. However, we assert that these ideas have merit in and of themselves, and also note that most have already been explored in limited form within the current GRAM protocol and/or the GARA prototype system.

Acknowledgments

We are grateful to many colleagues for discussions on the topics discussed here, in particular Larry Flon, Jeff Frey, Steve Graham, Bill Johnston, Miron Livny, Jeff Nick, and Alain Roy. This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the National Science Foundation; by the NASA Information Power Grid program; and by IBM.

References

[1] SOAP version 1.2 part 0: Primer. W3C Working Draft 17. www.w3.org/TR/-soap12-part0/. 160
[2] Jim Basney and Miron Livny. Managing network resources in Condor. In *Proc. 9th IEEE Symp. on High Performance Distributed Computing*, 2000. 172
[3] Michael Beynon, Renato Ferreira, Tahsin M. Kurc, Alan Sussman, and Joel H. Saltz. Datacutter: Middleware for filtering very large scientific datasets on archival storage systems. In *IEEE Symposium on Mass Storage Systems*, pages 119–134, 2000. 154
[4] J. Boyle, R. Cohen, D. Durham, S. Herzog, R. Rajan, and A. Sastry. The COPS (Common Open Policy Service) protocol. IETF RFC 2748, January 2000. 172
[5] S. Brunett, D. Davis, T. Gottschalk, P. Messina, and C. Kesselman. Implementing distributed synthetic forces simulations in metacomputing environments. In *Proceedings of the Heterogeneous Computing Workshop*, pages 29–42. IEEE Computer Society Press, 1998. 154, 173
[6] Sharon Brunett, Karl Czajkowski, Steven Fitzgerald, Ian Foster, Andrew Johnson, Carl Kesselman, Jason Leigh, and Steven Tuecke. Application experiences with the Globus toolkit. In *HPDC7*, pages 81–89, 1998. 169, 173
[7] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. Technical report, W3C, 2001. http://www.w3.org/TR/wsdl/. 160

[8] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998. 155, 169, 171, 173

[9] Karl Czajkowski, Alper K. Demir, Carl Kesselman, and M. Thiebaux. Practical resource management for grid-based visual exploration. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 2001. 154, 159, 173

[10] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid information services for distributed resource sharing. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 2001. 155, 169

[11] Karl Czajkowski, Ian Foster, and Carl Kesselman. Co-allocation services for computational grids. In *Proc. 8th IEEE Symp. on High Performance Distributed Computing*. IEEE Computer Society Press, 1999. 171, 173

[12] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the internet. *ACM/Springer Verlag Journal on Multimedia Systems*, 5(3), 1997. 172

[13] D. Draper, P. Fankhauser, M. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler, editors. *XQuery 1.0 Formal Semantics*. W3C, March 2002. http://www.w3.org/TR/2002/WD-query-semantics-20020326/. 177

[14] D. C. Fallside. XML schema part 0: Primer. Technical report, W3C, 2001. http://www.w3.org/TR/xmlschema-0/. 171

[15] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. *ACM/Springer Verlag Journal on Multimedia Systems*, 5(3), 1997. 172

[16] I. Foster and C. Kesselman. The Globus project: A status report. In *Proceedings of the Heterogeneous Computing Workshop*, pages 4–18. IEEE Computer Society Press, 1998. 169

[17] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999. 153, 175

[18] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. Technical report, Globus Project, 2002. www.globus.org/research/papers/ogsa.pdf. 155, 159, 168

[19] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computers and Security*, pages 83–91. ACM Press, 1998. 155, 168

[20] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *Intl. Journal of High Performance Computing Applications*, 15(3):200–222, 2001. http://www.globus.org/research/papers/-anatomy.pdf. 153, 159

[21] I. Foster, A. Roy, and V. Sander. A Quality of Service Architecture that Combines Resource Reservation and Application Adaptation. In *International Workshop on Quality of Service*, 2000. 155, 169, 171, 173

[22] I. Foster, A. Roy, V. Sander, and L. Winkler. End-to-End Quality of Service for High-End Applications. Technical report, Argonne National Laboratory, Argonne, 1999. http://www.mcs.anl.gov/qos/qos_papers.htm. 154, 155, 169, 171, 173

[23] Roch Guérin and Henning Schulzrinne. Network quality of service. In *[17]*, pages 479–503. 172

[24] A. Hafid, G. Bochmann, and R. Dssouli. A quality of service negotiation approach with future reservations (nafur): A detailed study. *Computer Networks and ISDN Systems*, 30(8), 1998. 172

[25] Hao hua Chu and Klara Nahrstedt. CPU service classes for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 296–301. IEEE Computer Society Press, June 1999. Florence, Italy. 172

[26] Tahsin Kurc, Ümit Çatalyürek, Chialin Chang, Alan Sussman, and Joel Salz. Exploration and visualization of very large datasets with the Active Data Repository. Technical Report CS-TR-4208, University of Maryland, 2001. 154, 159

[27] M. Livny. Matchmaking: Distributed resource management for high throughput computing. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, 1998. 171

[28] A. Mehra, A. Indiresan, and K. Shin. Structuring communication software for quality-of-service guarantees. In *Proc. of 17th Real-Time Systems Symposium*, December 1996. 172

[29] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997. 177

[30] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware resource management for distributed multimedia applications. *Journal on High-Speed Networking, IOS Press*, December 1998. 172

[31] K. Nahrstedt and J. M. Smith. Design, implementation and experiences of the OMEGA end-point architecture. *IEEE JSAC, Special Issue on Distributed Multimedia Systems and Technology*, 14(7):1263–1279, September 1996. 172

[32] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *The IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, June 2002. 158

[33] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981. 177

[34] Rajesh Raman, Miron Livny, and Marvin Solomon. Resource management through multilateral matchmaking. In *Proc. 9th IEEE Symp. on High Performance Distributed Computing*, 2000. 171

[35] L. Rodrigues, K. Guo, P. Verissimo, and K. Birman. A dynamic light-weight group service. *Journal on Parallel and Distributed Computing*, (60):1449–1479, 2000. 169

[36] V. Sander, W. A. Adamson, I. Foster, and A. Roy. End-to-End Provision of Policy Information for Network QoS. In *Proc. 10th IEEE Symp. on High Performance Distributed Computing*, 2001. 155, 169, 173

[37] P. Stelling, I. Foster, C. Kesselman, C. Lee, and G. von Laszewski. A fault detection service for wide area distributed computations. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 268–278, 1998. 163

[38] B. Teitelbaum, S. Hares, L. Dunn, V. Narayan, R. Neilson, and F. Reichmeyer. Internet2 QBone - Building a testbed for differentiated services. *IEEE Network*, 13(5), 1999. 172

[39] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, and C. Kesselman. Grid services specification. Technical report, Globus Project, 2002. www.globus.org/-research/papers/gsspec.pdf. 155

[40] J. Vollbrecht, P. Calhoun, S. Farrell, L. Gommans, G. Gross, B. de Bruijn, C. de Laat, M. Holdrege, and D. Spence. AAA authorization application examples. Internet RFC 2905, August 2000. 168
[41] Gregor von Laszewski, Ian Foster, Joseph A. Insley, John Bresnahan, Carl Kesselman, Mei Su, Marcus Thiebaux, Mark L. Rivers, Ian McNulty, Brian Tieman, and Steve Wang. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1999. 154
[42] L. C. Wolf and R. Steinmetz. Concepts for reservation in advance. *Kluwer Journal on Multimedia Tools and Applications*, 4(3), May 1997. 172
[43] Ikjun Yeom and A. L. Narasimha Reddy. Modeling TCP behavior in a differentiated-services network. Technical report, TAMU ECE, 1999. 172

A SNAP Operational Semantics

Below, we provide a formal specification of the behavior of SNAP managers in response to agreement protocol messages. This might be useful to validate the behavior of an implementation, or to derive a model of the client's *belief* in the state of a negotiation. We use a variant of structural operational semantics (SOS) which is commonly used to illustrate the transformational behavior of a complex system [33, 29, 13]. We define our own system configuration model to retain as intuitive a view of protocol messaging as possible.

Our SOS may appear limited in that it only models the establishment of explicit SLAs without capturing the implicitly-created SLAs mentioned in Section 3.3. We think these implicit SLAs should not be part of a standard, interoperable SNAP protocol model, though a particular implementation might expose them. There are four main parts to our SOS:

1. Agreement language. An important component of the semantics captures the syntax of a single agreement, embedding the resource language from Section 4.
2. Configuration language. The state of a negotiation that is evolving due to manager state and messages between clients and manager.
3. Service functions. A set of function declarations that abstract complex decision points in the SNAP model. These functions are characterized but not exactly defined, since they are meant to isolate the formal model from implementation details.
4. Transition rules. Inference rules showing how the configuration of a negotiation evolves under the influence of the service predicates and the passage of time.

This SOS is not sufficient to understand SNAP behavior and SLA meaning until a concrete language is specified to support the $\mathbf{R} \subseteq \mathbf{J}$ languages proposed above.

A.1 Agreement Language

An agreement a appears in the SLA language A, a generic 4-tuple as introduced in Section 3.2:

$$d \in \mathrm{D} = \langle \mathbf{R} \rangle_{\mathrm{R}} + \langle \mathbf{J} \rangle_{\mathrm{T}} + \langle \mathbf{J} \rangle_{\mathrm{B}} + \epsilon$$
$$a \in \mathrm{A} = \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{D}$$

The domain D of SLA *descriptions* is a union of the individual descriptive languages described in Section 4. Because these descriptions share the same $\mathbf{R} \subset \mathbf{J}$ language, we wrap them with type designation to distinguish the content of RSLA, TSLA, and BSLA descriptions. An SLA containing the special ϵ-description represents an identifier which is allocated but not yet associated with SLA terms. Additional terminal domains I, N, and T are assumed for identifiers, client names, and time values, respectively.

A.2 Configuration Model

Abstractly, a configuration of negotiation between clients and a manager is a tuple of an input message queue Q, the agreement state A of the manager, an output message set X, and the manager's clock t:

$$\langle Q, A, X, t \rangle$$

The syntax of the configuration is specified as follows using a mixture of BNF grammar and domain-constructors:

$$\begin{aligned} q \in \mathrm{M_{in}} := & \ \mathbf{getident}(c, t) \\ | & \ \mathbf{setdeath}(I, c, t) \\ | & \ \mathbf{request}(I, c, t, d) \\ | & \ \mathbf{clock}(t) \end{aligned}$$

$$\begin{aligned} \mathrm{M_{out}} := & \ \mathbf{useident}(I, c, t) \\ | & \ \mathbf{willdie}(I, c, t) \\ | & \ \mathbf{agree}(I, c, t, d) \\ | & \ \mathbf{error}() \end{aligned}$$

$$\langle Q, A, X, t \rangle \ \in \ \mathrm{M_{in}^{*}} \times \mathcal{P}(\mathrm{A}) \times \mathcal{P}(\mathrm{M_{out}}) \times \mathrm{T}$$

For the benefit of the following SOS rules, we include client identifiers in the message signatures which were omitted from the messages when presented in Section 3.

A.3 Service Functions

This formulation depends on a number of abstractions to isolate the implementation or policy-specific behavior of a SNAP manager. The following support functions are described in terms of their minimal behavioral constraints, without suggesting a particular implementation strategy.

Set Manipulation We use polymorphic set operators + and − to add and remove distinct elements from a set, respectively:

$$+ : \mathcal{P}(\tau) \times \tau \rightarrow \mathcal{P}(\tau)$$
$$= \lambda\, S, v \,.\, S \cup \{v\}$$

$$- : \mathcal{P}(\tau) \times \tau \rightarrow \mathcal{P}(\tau)$$
$$= \lambda\, S, v \,.\, \{x \,|\, x \in S \wedge x \neq v\}$$

Requirements Satisfaction As discussed in Sections 3.2 and 5, we assume a relation $\sqsubseteq$ between descriptions indicating how their solution spaces are related:

$$\sqsubseteq : \mathrm{R} \times \mathrm{R} \rightarrow \mathrm{Bool}$$
$$\sqsubseteq : \mathrm{J} \times \mathrm{J} \rightarrow \mathrm{Bool}$$

Basic Services

Function authz maps a client name to a truth value, yielding true if and only if the client is authorized to participate in SNAP negotiations:

$$\mathrm{authz} : \mathrm{N} \rightarrow \mathrm{Bool}$$

Function newident provides a new identifier that is distinct from all identifiers in the input agreement set:

$$\mathrm{newident} : \mathrm{A} \rightarrow \mathrm{I}$$
$$= \lambda\, A \,.\, i \,|\, \langle i, \ldots \rangle \notin A$$

Initial Agreement The "reserve," "schedule," and "bind" functions choose a new SLA to satisfy the client's request, or ⊥ (bottom) if the manager will not satisfy the request.

Function reserve chooses a new RSLA:

$$\mathrm{reserve} : \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{R} \rightarrow \mathrm{A}$$
$$= \lambda\, A, I, c, t, r \,.\, \begin{cases} \langle I, c, t, \langle r' \rangle_{\mathrm{R}} \rangle \,|\, r' \sqsubseteq r \\ \bot \end{cases}$$

Function schedule chooses a new TSLA:

$$\mathrm{schedule} : \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{J} \rightarrow \mathrm{A}$$
$$= \lambda\, A, I, c, t, j \,.\, \begin{cases} \langle I, c, t, \langle j' \rangle_{\mathrm{T}} \rangle \,|\, j' \sqsubseteq j \\ \bot \end{cases}$$

Function bind chooses a new BSLA:

$$\begin{aligned}\text{bind} &: \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{J} \rightarrow \mathrm{A} \\ &= \lambda\, A, I, c, t, j\,.\begin{cases} \langle I, c, t, \langle j' \rangle_{\mathrm{B}} \rangle \mid j' \sqsubseteq j \\ \bot \end{cases}\end{aligned}$$

Change Agreement The "rereserve," "reschedule," and "rebind" functions choose a replacement SLA to satisfy the client's request as discussed in Section 3.4, or $\bot$ if the manager will not satisfy the request.

Function rereserve chooses a replacement RSLA:

$$\begin{aligned}\text{rereserve} &: \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{R} \rightarrow \mathrm{A} \\ &= \lambda\, A, I, c, t, r\,.\begin{cases} \langle I, c, t, \langle r' \rangle_{\mathrm{R}} \rangle \mid r' \sqsubseteq r \\ \bot \end{cases}\end{aligned}$$

Function reschedule chooses a replacement TSLA:

$$\begin{aligned}\text{reschedule} &: \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{J} \rightarrow \mathrm{A} \\ &= \lambda\, A, I, c, t, j\,.\begin{cases} \langle I, c, t, \langle j' \rangle_{\mathrm{T}} \rangle \mid j' \sqsubseteq j \\ \bot \end{cases}\end{aligned}$$

Function rebind chooses a replacement BSLA:

$$\begin{aligned}\text{rebind} &: \mathrm{A} \times \mathrm{I} \times \mathrm{N} \times \mathrm{T} \times \mathrm{J} \rightarrow \mathrm{A} \\ &= \lambda\, A, I, c, t, j\,.\begin{cases} \langle I, c, t, \langle j' \rangle_{\mathrm{B}} \rangle \mid j' \sqsubseteq j \\ \bot \end{cases}\end{aligned}$$

A.4 Transition Rules

The following transitions rules serve to describe how a SNAP configuration of manager SLA set and message environment evolves during and after negotiation. Input messages are processed according to these rules to change the SLA set of the manager and to issue response messages. Each transition is structured as an inference rule with a number of antecedent clauses followed by a consequent rewrite of the SNAP configuration:

$$\frac{\begin{array}{l}\text{antecedent}_1 \\ \ldots\end{array}}{\langle q.Q, A, X, t \rangle \Rightarrow \langle Q, A', X', t' \rangle}$$

The first matching rule is used to rewrite the configuration.

Lifetime Management

New identifiers are allocated as needed:

$$\frac{\begin{array}{l}\text{authz}(c)\\ t_0 < t_1\\ I = \text{newident}(A)\\ a = \langle I, c, t_1, \epsilon\rangle\end{array}}{\langle \textbf{getident}(c, t_1).Q, A, X, t_0\rangle \Rightarrow \langle Q, A + a, X + \textbf{useident}(I, c, t_1), t_0\rangle}$$

Timeout changes affect existing agreements:

$$\frac{\begin{array}{l}a_1 = \langle I, c, t_1, \ldots\rangle \in A\\ a_2 = \langle I, c, t_2, \ldots\rangle\\ A' = A - a_1 + a_2\end{array}}{\langle \textbf{setdeath}(I, c, t_2).Q, A, X, t_0\rangle \Rightarrow \langle Q, A', X + \textbf{willdie}(I, c, t_2), t_0\rangle}$$

Clock advances trigger removal of stale agreements:

$$\frac{\begin{array}{l}t_0 < t_1\\ A' = \{\langle I, c, t, \ldots\rangle \,|\, \langle I, c, t, \ldots\rangle \in A \;\wedge t > t_1\}\end{array}}{\langle \textbf{clock}(t_1).Q, A, X, t_0\rangle \Rightarrow \langle Q, A', X, t_1\rangle}$$

The clock message is not originated by clients, but rather synthesized within the implementation. It is formalized as a message to capture the isochronous transition semantics of the manager state with regard to messages and the passing time.

Initial Agreement A new agreement is considered when a client requests an agreement on a stub identifier agreement.

New RSLA

$$\frac{\begin{array}{l}t_0 < t_2\\ a_1 = \langle I, c, t_1, \epsilon\rangle \in A\\ a_2 = \langle I, c, t_2, \langle r'\rangle_{\text{R}}\rangle = \text{reserve}(A, I, c, t_2, r)\\ r' \sqsubseteq r\\ A' = A - a_1 + a_2\end{array}}{\langle \textbf{reqest}(I, c, t_2, \langle r\rangle_{\text{R}}).Q, A, X, t_0\rangle \Rightarrow \langle Q, A', X + \textbf{agree}(I, c, t_2, \langle r'\rangle_{\text{R}}), t_0\rangle}$$

New TSLA

$$\frac{\begin{array}{l}t_0 < t_2\\ a_1 = \langle I, c, t_1, \epsilon\rangle \in A\\ a_2 = \langle I, c, t_2, \langle j'\rangle_{\text{T}}\rangle = \text{schedule}(A, I, c, t_2, j)\\ j' \sqsubseteq j\\ A' = A - a_1 + a_2\end{array}}{\langle \textbf{reqest}(I, c, t_2, \langle j\rangle_{\text{T}}).Q, A, X, t_0\rangle \Rightarrow \langle Q, A', X + \textbf{agree}(I, c, t_2, \langle j'\rangle_{\text{T}}), t_0\rangle}$$

New BSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \epsilon \rangle \in A \\ a_2 = \langle I, c, t_2, \langle j' \rangle_{\mathrm{B}} \rangle = \mathrm{bind}(A, I, c, t_2, j) \\ j' \sqsubseteq j \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle j \rangle_{\mathrm{B}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle j' \rangle_{\mathrm{B}}), t_0 \rangle}$$

Repeat Agreement If a client requests an agreement on an existing agreement, and the existing agreement already satisfies the request, then a repeat acknowledgment is sent and the termination time of the existing agreement is adjusted to the current request.

Repeat RSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle r' \rangle_{\mathrm{R}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle r' \rangle_{\mathrm{R}} \rangle \\ r' \sqsubseteq r \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle r \rangle_{\mathrm{R}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle r' \rangle_{\mathrm{R}}), t_0 \rangle}$$

Repeat TSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle j' \rangle_{\mathrm{T}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle j' \rangle_{\mathrm{T}} \rangle \\ j' \sqsubseteq j \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle j \rangle_{\mathrm{T}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle j' \rangle_{\mathrm{T}}), t_0 \rangle}$$

Repeat BSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle j' \rangle_{\mathrm{B}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle j' \rangle_{\mathrm{B}} \rangle \\ j' \sqsubseteq j \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle j \rangle_{\mathrm{B}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle j' \rangle_{\mathrm{B}}), t_0 \rangle}$$

Change Agreement If a client requests an agreement on an existing agreement of the same type, but the existing agreement does not satisfy the request, an SLA change is considered.

Change RSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle r' \rangle_{\mathrm{R}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle r' \rangle_{\mathrm{R}} \rangle = \mathrm{rereserve}(A, I, c, t_2, r) \\ r' \sqsubseteq r \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle r \rangle_{\mathrm{R}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle r' \rangle_{\mathrm{R}}), t_0 \rangle}$$

Change TSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle j' \rangle_{\mathrm{T}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle j' \rangle_{\mathrm{T}} \rangle = \mathrm{reschedule}(A, I, c, t_2, j) \\ j' \sqsubseteq j \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle j \rangle_{\mathrm{T}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle j' \rangle_{\mathrm{T}}), t_0 \rangle}$$

Change BSLA

$$\frac{\begin{array}{l} t_0 < t_2 \\ a_1 = \langle I, c, t_1, \langle j' \rangle_{\mathrm{B}} \rangle \in A \\ a_2 = \langle I, c, t_2, \langle j' \rangle_{\mathrm{B}} \rangle = \mathrm{rebind}(A, I, c, t_2, j) \\ j' \sqsubseteq j \\ A' = A - a_1 + a_2 \end{array}}{\langle \mathbf{reqest}(I, c, t_2, \langle j \rangle_{\mathrm{B}}).Q, A, X, t_0 \rangle \Rightarrow \langle Q, A', X + \mathbf{agree}(I, c, t_2, \langle j' \rangle_{\mathrm{B}}), t_0 \rangle}$$

Error Clause If none of the above inference rules match, this one signals an error to the client. A quality implementation would provide more elaborate error signaling content.

$$\frac{}{\langle q.Q, A, X, t \rangle \Rightarrow \langle Q, A, X + \mathbf{error}(), t \rangle}$$