SHARP: An Architecture for Secure Resource Peering*

Yun Fu[†], Jeffrey Chase[†], Brent Chun[‡], Stephen Schwab[§] and Amin Vahdat[†]

ABSTRACT

This paper presents SHARP, a framework for secure distributed resource management in an Internet-scale computing infrastructure. The cornerstone of SHARP is a construct to represent cryptographically protected resource *claims*—promises or rights to control resources for designated time intervals—together with secure mechanisms to subdivide and delegate claims across a network of resource managers. These mechanisms enable flexible *resource peering*: sites may trade their resources with peering partners or contribute them to a federation according to local policies. A separation of claims into *tickets* and *leases* allows coordinated resource management across the system while preserving site autonomy and local control over resources. SHARP also introduces mechanisms for controlled, accountable *oversubscription* of resource claims as a fundamental tool for dependable, efficient resource management. We present experimental results from a SHARP prototype for PlanetLab, and illustrate its use with a decentralized barter economy for global PlanetLab resources. The results demonstrate the power and practicality of the architecture, and the effectiveness of oversubscription for protecting resource availability in the presence of failures.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.6 [**Operating Systems**]: Security and Protection; H.4.3 [**Information Systems Applications**]: Communications Applications

*This research is supported in part by the National Science Foundation (EIA-99772879 and ITR-0082912), Hewlett Packard, IBM, and Intel. Vahdat is also supported by an NSF CAREER award (CCR-9984328).

[†]Department of Computer Science, Duke University, {*fu,chase,vahdat*}*@cs.duke.edu*.

[‡]Intel Research Berkeley, *bnc@intel-research.net*.

[§]Network Associates Laboratories, *sschwab@nai.com*.

SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

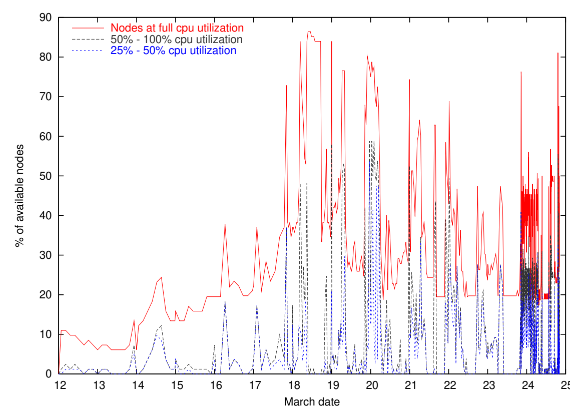

Figure 1: Demand for PlanetLab production nodes leading up to the SOSP paper submission deadline (March 24, 2003). The heavy, bursty demand made it difficult for authors to find resources or obtain stable results. Other resources (e.g., network bandwidth) showed similar impacts.

General Terms

Experimentation, Management, Performance, Security

Keywords

Resource Allocation, Resource Peering, Peer-to-peer

1. INTRODUCTION

Several research threads are converging toward federated sharing of dispersed pools of networked computing resources under coordinated control. Examples include Internet service utilities (e.g., Content Services Networks), computational network overlays such as PlanetLab [36] and Netbed [48], peer-to-peer services, and grid computing systems, which harness federated computing resources for massive computational problems and network services [23]. All of these systems are built above rapidly maturing support for *location-independent* service naming and instantiation.

These systems need effective resource management for fair sharing of community resources, performance isolation and predictability, and adaptivity to changing conditions. As one motivating example, Figure 1 shows a classic "tragedy of the commons" for PlanetLab during a period of high demand. Here, a growing number of PlanetLab users simultaneously request "slices" of resources from arbitrarily selected nodes to host distributed systems experiments. In this system, the PlanetLab nodes schedule their requests locally, with no mechanism to discover or reserve resources, coordinate resource usage across the system, or control resource usage by users or groups. Users have little basis to predict the resources available to them at each site, creating an incentive to request more resources than needed. Users who obtain poor results due to overloading at one or more sites either retry their experiments—consuming even more resources—or give up. The scenario is similar to congestion collapse in the Internet.

This paper proposes a new approach to flexible resource management for wide-area networked systems such as PlanetLab. Consider a collection of logical *sites* or domains, each running local schedulers for physical resources (e.g., processors, memory, storage, network links, sensors) under its control. A site may be as small as a single computer, or it could be a large group of resources under common ownership, analogous to an autonomous system. While the resources within each site may be highly dynamic, we assume that the sites themselves are reasonably long-lived and static. The actors in the system are software programs operating on behalf of users and organizations, which export local resources and consume global resources. The goal of our work is to develop fundamental abstractions and mechanisms to allocate resources across the system in a coordinated way, under the direction of pluggable policies for discovering resources, matching requests to available resources, and assigning priority or control over resources.

A system for flexible policy-based resource management must meet several basic goals. It must allow actors to reserve resources across the system for predictable behavior, and it must prevent actors from stealing resources held by others. It must support admission control, so that users have an opportunity to abort or redirect resource requests that cannot be met in full, without consuming resources unnecessarily. It must balance global resource sharing with local autonomy, leaving sites ultimate control over their resources and the flexibility to adjust use of those resources to respond to local conditions. The system must be robust: in particular, it must protect resource availability if a resource holder fails or becomes unreachable. Finally, it must be secure: the actors may be mutually distrusting and are subject to attack and subversion from third parties.

We could address many of these goals with a trusted central resource manager that assumes ownership of all resources, authenticates itself to the sites, and directs their local schedulers remotely. Our approach establishes a more decentralized structure in which multiple resource managers (brokers or *agents*) control different, possibly overlapping portions of the global resource pool. It is based on mechanisms to represent resource *claims*—promises or rights to possess resources—and to subdivide and transfer or delegate claims in a way that is cryptographically verifiable by third parties. Secure claim delegation makes it possible to add new agents to the system and assign them control over bundles of resources spanning any number of sites. It creates an open framework in which multiple resource management policies may coexist to serve different application classes and different segments of the user community, with no central point of trust, failure or attack. More generally, it creates a necessary foundation for resource management based on peering, bartering, or economic exchange—a secure computational economy [45, 49]. For example site A may grant to site B (or an agent serving site B) a claim on resources at A in exchange for a claim to like access at B, possibly at a different time, or other compensation.

This paper presents the design and implementation of SHARP (*Secure Highly Available Resource Peering*), a new architecture for distributed resource management, resource control, and resource sharing across sites and trust domains. A key goal of SHARP is to reconcile policy-based resource management with resource availability when agents or other actors fail, become unreachable, or abandon their claims. SHARP is based on soft-state *timed claims* that expire after a specified period, so the system can recover the resources if a claim holder fails, following the classical *lease* model [25]. Also, agents may *oversubscribe* resources to improve resource efficiency and availability when claims are lost or left idle; claim holders have *probabilistic* assurance that their claims will be honored. Sites can detect oversubscribed claims and may reject them to prevent principals or their delegates from claiming more than their allotted share of resources. Claim delegation is *accountable* to protect users against fraudulent agents.

SHARP claims are cryptographically signed to make them unforgeable and non-repudiable. Our architecture avoids any assumption of a trusted certification authority, eliminating global key management as a deployment obstacle. In particular, each site is free to act as its own authority to certify keys and to grant or validate claims on its local resources. Each claim is authorized by a chain of signed delegations anchored in the site authority itself [27, 50, 9]. SHARP claims are *self-certifying*: an agent endorses the keys of its peering partners after validating them using any locally preferred authentication mechanism [32]. In essence, each site is the root of a certificate mesh that parallels the resource peering relationships, certifying transitive trust for all principals that claim its resources.

We present experimental results from a SHARP prototype using an XML-based resource peering protocol for PlanetLab Slices [36]. The results demonstrate resource management scenarios running across PlanetLab, including use of oversubscribed claims to control flexible tradeoffs between resource efficiency, resource availability, and claim rejection rates. To illustrate the power of the SHARP framework we describe and evaluate a simple system for global resource trading and resource discovery based on pair-wise barter exchanges. Bartering enables transparent access to global resources without central agreement; a site may join the federation simply by establishing a peering relationship with any member site. This approach illustrates one way that SHARP can remove barriers to joining federated server networks such as PlanetLab.

Section 2 gives an overview of the SHARP architecture. Section 3 defines mechanisms for secure claims and resource delegation, and presents a security analysis. Section 4 discusses the role of soft claims and oversubscription in highly available distributed resource management. Section 5 presents experimental results from a SHARP prototype for PlanetLab, using a barter economy as a case study. Section 6 summarizes related work, and Section 7 concludes.

2. OVERVIEW

Figure 2 illustrates the actors in the SHARP framework and their interactions. Each actor runs on behalf of one or

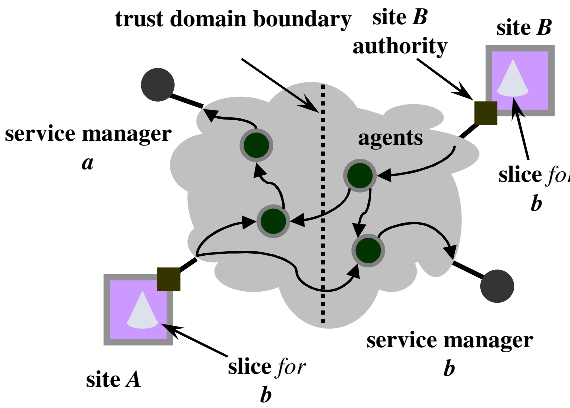

Figure 2: Resource peering across sites. Sites delegate control over their resources to agents; service managers contact agents to obtain resources to run a service. Here, an agent grants service manager b a slice of computing resources spanning sites A and B, enabling b to instantiate its service at both sites.

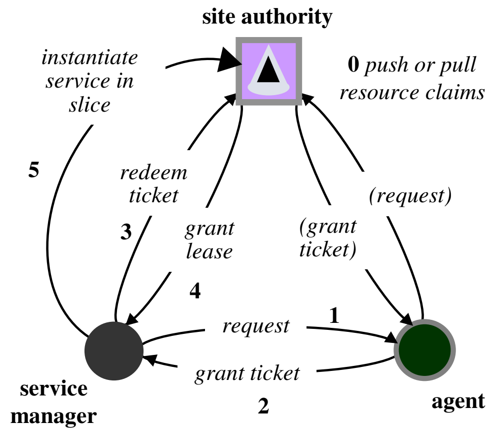

Figure 3: An exchange between a site authority, agent, and service manager. The service manager requests a ticket from an agent, redeems it with the site authority for a lease on a slice of the site's resources, and instantiates a guest service in the slice.

more principals and possesses at least one asymmetric key pair. In a typical use of Sharp to manage a shared server network the consumers of resources are network application services hosted in the system. Each program runs within a *slice*—a partition or share of global resources assigned to it. A *service manager* for each guest service is responsible for obtaining resources, binding them to the slice, and instantiating the service within the slice.

The resources at each site are controlled by a *site authority*, which maintains hard state about resource status and slices at the site, participates in Sharp exchanges to export the site's resources, and handles claims to allocate resources at the site. A local resource scheduler is responsible for enforcing the slice assignments within each site [6, 44, 21, 36]. For example, each PlanetLab site consists of a single node with a virtual machine monitor that assigns *slivers* of its resources to virtual machines using local scheduling mechanisms similar to Resource Containers [6] or VMware ESX [44], as described in Section 5; a PlanetLab slice is an aggregation of slivers on one or more nodes. A cluster manager that assigns blocks of servers could also act as a site authority for a server farm or utility data center [5, 14].

Resource managers called *Agents* mediate between site authorities and resource consumers. An agent is any entity that issues Sharp resource claims to another principal. Site authorities also act as agents when they issue claims to delegate control over their resources to external managers, e.g., to export local resources to a federation or grid according to local peering policies.

2.1 Resource Claims: Tickets and Leases

The key construct in Sharp is a *resource claim*, which allows principals to formulate and exchange unforgeable assertions about control over resources. A claim record is an XML object asserting that a specified principal (the *holder*) controls some resources (a *resource set*) over some time interval (its *term*). To protect the integrity of claims, each claim record is signed with the private key of the principal conferring ownership of the resource (the *issuer*). The issuer names the holder by a public key; by signing the transfer, the issuer certifies that it trusts any entity possessing the corresponding private key to control the resources named in the claim. Sections 3.1 and 3.2 discuss resource sets and claims in more detail.

Service managers obtain resources at a Sharp site using the two-phase process depicted in Figure 3. In the first phase, the service manager obtains a resource claim in the form of a *ticket* from an agent. A ticket represents a *soft* claim that suggests but does not guarantee resource ownership. Tickets support probabilistic resource allocation as described below. In the second phase, the service manager presents the ticket to the appropriate site authority to redeem it; the authority may reject the ticket or it may honor the request by issuing a *lease* for any subset of the resources or term specified in the ticket. A lease is a *hard* claim over concrete resources, and is guaranteed valid for its term unless a failure occurs. This distinction between tickets and leases is crucial to Sharp because it preserves local autonomy and local control over resources. Agents hold soft claims (tickets) instead of concrete reservations, and the system defers the binding to concrete resources so the authority may consider current conditions at the site in determining how to redeem the claim. This idea is analogous to seat bookings on airlines: the ticket promises a seat on a flight, but the holder must successfully redeem it for a boarding pass before boarding the flight.

A ticket holder may *delegate* a portion of its resources to another principal by issuing a ticket containing a new claim signed with its private key, as described in Section 3. The complete ticket comprises the sequence of claim records validating each step in the ticket's delegation chain; the final claim record in the sequence names the ticket's holder, resource set, and term. Ticket delegation is similar to a transfer of access privileges in capability-based systems [31], but tickets differ from capabilities in several key respects. For example, they are probabilistic, they expire, they represent abstract promises of resources rather than referencing specific objects, they contain a complete record of their delegations, and they are bound to the holder's key pair. Note that Sharp's domain mitigates the confinement problem traditionally associated with delegation of capabilities. For instance, a leaked capability in a capability-based operating system [2, 40] or file system may permanently compromise sensitive information or threaten the integrity of arbitrary programs. In SHARP, the damage from a leaked ticket is limited to the value of the named resources for the duration of the ticket's term. See Section 3.5 for a more complete discussion of this issue.

SHARP tickets are self-describing and self-certifying. The site authority or any third party can independently validate the delegation chain before granting a lease or other resources in exchange for the ticket. Even if the receiver has no previous trust in one or more entities in the chain or no knowledge of their public keys, it can verify that they have limited transitive trust to manage the resources for specified time intervals. In particular, the site authority can verify its own limited transitive trust in the ticket holder to control the resources named in the ticket—without contacting the agent that issued it. Agents may use any local mechanism to authenticate their delegates; while they are free to obtain certified keys from global key services (e.g., a Public Key Infrastructure), they are not required to do so, and the architecture avoids any assumption of global trust or global identity. Section 2.4 discusses the security architecture in more detail.

2.2 Probabilistic Claims

A SHARP agent may *oversubscribe* its resources by issuing more tickets than it can support from the resource claims that it holds. Oversubscribing may be a malicious act to "inflate" tickets, but it is also a valuable tool for dependable decentralized resource management. Oversubscribed claims can improve resource utilization by statistical multiplexing, and they support replicated tickets to limit the damage from resource loss if an agent fails or becomes unreachable. Section 4 explores the uses of oversubscribed tickets.

Oversubscription is safe in the sense that a site authority can identify and reject tickets that conflict with claims for which it has already granted leases. Thus oversubscribed tickets are *probabilistic*: the probability that a site authority will honor an oversubscribed ticket varies with the oversubscription degree and the rate of ticket redemption. Oversubscribed tickets in SHARP are distinct from related ideas in lottery-based resource management [46], which address similar goals. For instance, creating additional tickets in lottery scheduling leads to inflation, which reduces the value of each ticket. In SHARP, oversubscription does not affect a ticket's face value, but it makes it more likely that an authority will reject the ticket when the holder redeems it for a lease (see Section 6 for a more complete comparison).

SHARP tickets have the crucial property that delegation is *accountable*: given a set of tickets, it is efficient to determine if they are conflicting, and if so to identify an agent that issued an oversubscribed ticket and that can be held accountable for the conflict. To reject a ticket, the authority marks it as invalid, signs it, and returns it to the bearer, together with a set of previously redeemed claims identifying the accountable agent and proving its guilt. The victim may present the rejected claim set to the issuing agent for compensation, or to a third party to prove that the agent is responsible. This property makes it possible to employ reputation services [1, 29] or other mechanisms to adjudicate rejected claims, in order to remove any incentive for agents to violate their contracts. Section 3.4 discusses accountability further.

Request$\langle requestID, resourceSet, term, [claims], [optional]\rangle$
Request a ticket for the specified resource set and term. The optional claim set may encode a previously rejected ticket requiring compensation, a previously issued ticket to be extended, and/or tickets offered in trade for the requested resources. Optional arguments may substantiate the request with payment, certificates, or endorsements.

Claim$\langle requestID, claims\rangle$
Redeem one or more tickets for leases or lease extensions. The tickets are encoded in the claim set.

Grant$\langle requestID, claims\rangle$
Transfer tickets or leases, e.g., in response to an earlier request *requestID* that solicited the grant.

Reject$\langle requestID, rejectRecord, claims\rangle$
Reject a ticket presented in a *Claim* request *requestID*. The *rejectRecord* is a signed notice from the authority that rejected the claim. The claim set proves that the rejection is justified and identifies an accountable agent.

Table 1: Outline of the SHARP interfaces for exchanging claim sets encoding tickets and leases. All request handling is asynchronous. The unique *requestID* identifies the requester and a response binding.

Agents in the SHARP framework may combine resource allocation and resource discovery functions to varying degrees. Agents define their own policies to index their resource holdings, select resources to match each request, and exchange tickets with other agents. Each agent controls the oversubscription degree for the tickets that it issues against the resource claims that it holds. Freely oversubscribed tickets correspond to best-effort service or "hints"; this is similar to resource discovery systems in which resource brokers maintain approximate information about resources and their status, but do not control allocation (e.g., Globus GRIS [17]). In contrast, tickets with no oversubscription approximate hard reservations (in the absence of failure); in this case, obtaining a resource bundle from an agent is an *operation* rather than a *query*. The continuum between these extremes allows soft reservations with configurable degrees of assurance. Agents may also undersubscribe tickets to maintain a configurable reserve margin against resource failures.

2.3 Agents and Resource Peering

Table 1 outlines the key elements of the SHARP interface for discovering and exchanging resources. While the framework itself is policy-neutral, accountable ticket delegation through a network of agents creates a powerful, unified, and extensible framework for resource management. These agents can serve many roles for comprehensive wide-area resource management:

- **Site agents.** A site authority acts as an agent to distribute claims for site resources, e.g., to contribute resources to a federation or grid. Each site agent controls the share of local resources exposed externally by the scope of the tickets it issues. For example, sites may hold portions of their resources in reserve for local use, and/or distribute claims according to arbitrary peering policies, e.g., "A has access to 50% of B's resources on weekends". Sites may delegate management of their resources to other agents implementing specialized policies.

- **User agents.** Agents may serve specific user communities and control access based on user identity. In a federated system an agent for an organization might gather tickets for global resources and distribute them to the organization's users. In this case, the organization configures its application service managers to obtain resources from its agent; the agent may authenticate requests and assign tickets based on user priority or other local policies.

- **Brokers.** Agents may function as *brokers* that trade tickets from multiple sites but do not contribute or request resources themselves. Brokers may index diverse resources by attributes to match resources to high-level resource requirements [22, 37, 43]. For example, it may be useful to specify a resource set loosely by attributes, e.g., "two servers somewhere nearby with large memories". Brokers may propagate resource tickets and index them by their attributes as in distributed resource discovery frameworks [3, 17, 43].

- **Community banking.** A broker may function as a "bank" that accepts local tickets in exchange for tickets for resources at other sites. The bank could allow any site to join a federated resource sharing system by contributing tickets for its local resources.

- **Adaptive provisioning.** Application-specific agents and service managers may monitor guest service behavior and adjust resource slices and slice locations to meet service quality goals embodied in Service Level Agreements (SLAs) for end-to-end application performance [5, 13, 18, 20, 22]. The mechanisms to adapt slices by extending and renegotiating short-term claims are outside the scope of this paper.

Since they control the distribution of tickets, agents may implement any policy to schedule blocks of resources. For example, consider the PlanetLab scenario described in Section 1, in which a small community of research colleagues share the infrastructure on a self-policing first-come first-served basis. Agents can help users to locate suitable bundles of resources that are not already in use—wherever they exist—and obtain tickets to reserve those resources for predictable performance. When demand is bursty the agents may perform admission control and issue tickets for off-peak times.

In a larger user community, or one in which users are not mutually trusting, agents might implement arbitrary policies to control resource usage and prioritize under constraint. For example, agents in a resource economy could sell tickets, e.g., for micropayments, perhaps using a bidding or pricing scheme to set prices according to resource supply and demand. In a federated peering system, agents could distribute tickets in proportion to each site's contributions, leaving each site to control how it distributes the tickets among its local users. A wide range of resource management policies may coexist safely in agents governing different, possibly overlapping portions of the global resource pool—an important goal for large-scale testbeds in which resource management is itself a focus of research. Controlled oversubscription protects resource availability when agents fail or hold tickets for more resources than their clients can use. We leave a full exploration of policies for resource management and oversubscription to future research.

A1 Principals (represented by site authorities, agents, and service managers) are cryptographically bound to public keys by digitally signed certificates, including but not limited to self-certifying SHARP tickets. These certificate chains are rooted by a set of certificate authorities (CAs) trusted by the site authorities, including but not limited to the site authorities themselves.

A2 There is no single root CA or fixed set of predetermined and globally published root CAs.

A3 Each site or agent establishes out-of-band trust relationships with one or more sites or agents, including exchange and authentication of public keys using arbitrary external mechanisms, including but not limited to shared trust in an external CA, e.g., a Public Key Infrastructure (PKI).

A4 Each agent uses an external mechanism, including but not limited to shared trust in an external CA, to establish sufficient trust in a service manager or peer agent before delegating resources to it.

A5 Site authorities are implicitly authorized to control resources for the sites they manage. For example, they may obtain this authorization by executing a root-privilege daemon on a node.

A6 Sites have some external means to prevent abuse of legally obtained resources, e.g., by monitoring their use and evicting abusers.

A7 Service managers have some external means to verify that each site provides the contracted resources and that they function correctly, e.g., by monitoring the behavior of each guest service.

Table 2: Assumptions of the SHARP security architecture.

2.4 Security Architecture

Table 2 summarizes the assumptions of the SHARP security architecture. Principals are uniquely identified by public keys (A1), although each actor may hold many such keys. These keys may be created and exchanged using any locally approved mechanisms for generating and distributing keys (A1, A2) or authenticating trading partners across trust domains (A3). For example, keys within a trust domain are certified by one or more authorities; if an agent that serves users, service managers, and peer agents within the domain trusts these authorities to certify their keys then the agent may authenticate clients for identity-based access control or priority.

Table 3 enumerates the threat model for the SHARP security architecture. The purpose of the security mechanisms is to authenticate and authorize access to shared resources and enable each principal to detect when another is misbe-

- T1 Unauthorized service manager attempts to use (A) local site resources or (B) remote site resources
- T2 Man-in-the-middle attacker modifies or replays SHARP protocol messages or simultaneously delivers them to multiple receivers.
- T3 Unauthorized agent or client requests resources.
- T4 Site contributes faulty resources.
- T5 Attacker sends malformed requests or claims.
- T6 Attacker obtains resources by (A) replaying tickets or (B) simultaneously presenting tickets to multiple agents or site authorities.
- T7 Malicious (A) site authority or (B) agent falsely advertises tickets or leases for which resources do not exist.
- T8 Malicious site authority falsely rejects tickets.
- T9 Attacker abuses legally obtained resources for a denial-of-service attack by (A) requesting excess resources and leaving them idle or (B) using computing resources to attack a third party.

Table 3: Summary of threats and vulnerabilities.

- M1 (Grant) Authorization of service managers or agents to control specific resources by delegation of signed resource claims.
- M2 (Grant) Transitive delegation of resource claims chains of claims (tickets).
- M3 (Request or Grant) Intra-domain delegation of claims from site authorities or agents to principals within the same domain.
- M4 (Request or Grant) Cross-domain delegation of claims from site authorities or agents to external principals.
- M5 (Grant response) Local validation of a signatures in a self-certifying ticket chain.
- M6 (Reject response) Local validation of well-formed self-describing tickets (e.g., all claims are nested and valid for the requested interval).
- M7 (Reject response) Local detection of conflicting oversubscribed tickets at a site authority; local validation of conflicting claim set issued by a site authority to justify a ticket rejection.

Table 4: Summary of SHARP security mechanisms.

having. The architecture does not address confidentiality although it may be a side effect of other implementation choices.

The mechanisms for validating tickets and protecting the integrity of claims directly address threats T1-T6A. In these cases, a resource request is detectably fraudulent or conflicting, and the receiver may simply reject it. Threats T6B-T9 are more subtle forms of misbehavior. For example, an authority may fail to deliver the resources it has committed in its leases (T7A), or reject tickets without adequate justification (T8). An agent may oversubscribe tickets and refuse to compensate its victims (T7B, T6B). A service manager or slice may allocate resources and deny them to other actors, but fail to consume them (T9A). Ultimately, every system defends against this kind of detectable abuse of shared resources by identifying and controlling the offenders or ejecting them from the system entirely. For example, actors in a resource economy may subscribe to a service that assigns financial penalties or rates the reputations of other actors [1, 29]. The SHARP security architecture supports such social or economic policies: the non-repudiation and accountability properties of tickets allow an actor who discovers misbehavior to prove it to a third party. However, an agent in a resource economy may choose to certify an anonymous principal to control resources if it offers valid payment or barter; this principal is not bound to a fixed identity subject to external control. We defer a discussion of this issue and the related problems of confinement and sybil attacks [19] to the more detailed security analysis in Section 3.5.

3. SECURE RESOURCE DELEGATION

This section presents the design of the basic SHARP mechanisms: how to represent and process resource claims, delegate securely, ensure accountability and nonrepudiability for all parties, and validate claims presented for redemption. Table 4 summarizes the security mechanisms presented in this section. Section 3.5 outlines a security analysis with respect to the threat model.

3.1 Resource Sets

SHARP principals interact by requesting and trading claims for resource sets. In this paper we limit our focus to resource sets made up of discrete units of networked computing resources of a specified type. These units could be servers or virtual servers of a given power, instruments or sensors, storage lots, or any other enumerable commodity whose instances at each site are interchangeable. There may be many resource types with different attributes. For example, a resource set might specify a bundle of shares of multiple resources at a site (e.g., CPU shares, network and/or storage bandwidth) [20].

A claim in a ticket specifies an *abstract* resource set *rset*, which is a pair (*type*, *count*). These resources are controlled by site authorities, who issue claims for blocks of resource units to other actors, e.g., agents. The agents may subdivide their claims arbitrarily to distribute them to their clients, which may be other agents. The holder of an abstract claim may redeem it to the named site authority for a lease on a *concrete* resource set, e.g., a list of IP addresses for allocated servers or virtual machines. The site authorities control this mapping from abstract to concrete resources.

3.2 Resource Claims

A claim record is a contract asserting that the issuing principal *issuer* delegates control over resource set *rset* to the delegated principal *holder* for term *term*, and is signed by the issuer's private key. A claim c is *active* at time t if $c.term.start \leq t \leq c.term.end$. A claim c is *expired* if the current time $t > c.term.end$. Table 5 shows a simplified XML claim representation for PlanetLab.

Each SHARP claim also has a globally unique serial num-

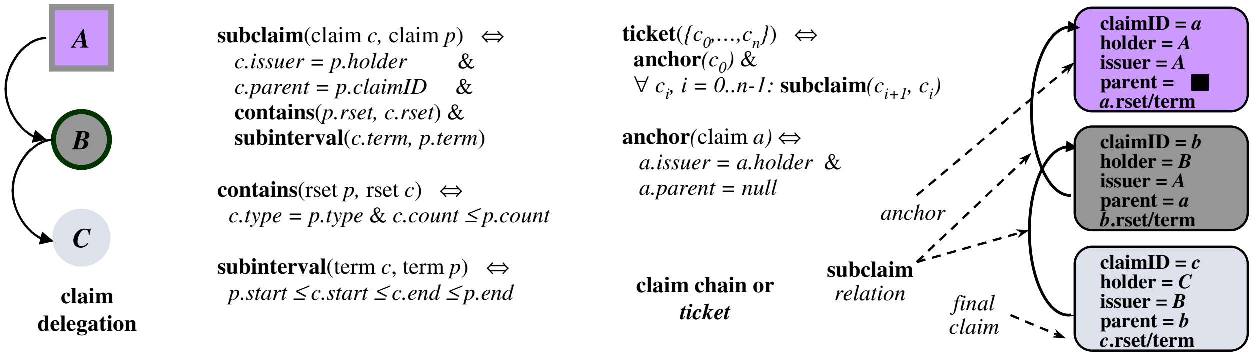

Figure 4: Conditions for valid subclaims and tickets, and a ticket comprising delegated claims from a site authority to a resource consumer through a broker/agent.

```
<subticket>
   <issuer> A's public key </issuer>
   <content>
      <principal>
        B's public key
      </principal>
      <nodes>
         number of virtual nodes (slivers)
      </nodes>
      <start_time> ... </start_time>
      <end_time> ... </end_time>
   </content>
   <signature>A's signature</signature>
</subticket>
```

Table 5: Simplified XML claim representation in PlanetLab. This claim record is one component or "subticket" of a complete ticket. The PlanetLab SHARP implementation is self-certifying and uses each principal's public key as a unique ID.

ber (*claimID*) and includes the serial number of a *parent* claim, as described below. Principals may track sets of claims by storing them in a local cache hashed by *claimID*. Each principal marks its claims with locally unique serial numbers; the *claimID* is qualified by the principal's unique ID to make it globally unique. The issuer signs the claim record by appending a secure hash (e.g., SHA) of the claim's contents encrypted with its private key K_i. Thus a claim record is a tuple:

$$claim = \langle claimID, issuer, holder, rset, term, parent\rangle_{SHA_{K_i}}$$

The signature protects the integrity of claims so that they may be arbitrarily cached, stored, transmitted through untrusted intermediaries, and replicated on untrusted servers. Claims are also *non-repudiable*; a principal cannot deny that it issued a claim. The signed *holder* prevents interception and hijacking of resource claims in transit. The unique *claimID* ensures that all copies of a claim are recognized as identical, preventing replay attacks. For example, site authorities may ensure that each claim is redeemed at most once by checking incoming tickets against a cache of previously redeemed claims that are still active (see Section 3.4).

3.3 Secure Delegation and Tickets

A holder of a claim p may delegate some or all of the claimed resources to another principal by generating a new claim c that is a *subclaim* of p. Figure 4 depicts delegated claims and the conditions for the **subclaim** predicate.

A claim c *descends* from p if it is a (transitive) subclaim of p. An immediate subclaim of p is a *child* of p. If p is not a child of any other claim, then $c.issuer$ is a site authority that directly controls the claimed resources: we refer to p as an *anchor* claim.

A *ticket* represents a claim substantiated by its chain of ancestor claims. Formally, a ticket is a sequence or *chain* of claim records $T = \{a, ..., f\}$ representing transitive subclaim delegations from an anchor claim a to the ticket's *final claim* f (refer to Figure 4). The ticket's final claim f specifies the ticket's holder, resource set, and term. The claim set making up a ticket constitutes a self-describing cryptographically secure proof that the authority $a.issuer$ has transitively delegated control over the resource set $f.rset$ to the principal $f.holder$ for term $f.term$. A ticket is *active* at time t if its final claim is active at time t. A ticket is *expired* if its final claim is expired.

Relationships among claims are securely apparent from inspection of a claim set. Once the claims are installed in a local cache hashed by *claimID* it is easy to traverse the delegation chain upwards from any claim by hashing on *parent* fields. Thus we refer to a cached set of claims descending from a common ancestor as a *claim tree*. A ticket T *descends* from a claim p if $c \in T$. It is easy to regenerate the complete ticket for any cached claim by tracing the path up the claim tree to its anchor. Figure 5 illustrates a set of hierarchical claims and the corresponding claim tree.

3.4 Ticket Conflicts and Accountability

SHARP must prevent any principal (or its delegates) from obtaining more of any site's resource than was transitively delegated to it, by identifying and rejecting tickets with conflicting resource claims. The site authority fills this role as a central point of ticket redemption for each site.

A set of claims $\{c_0, ..., c_n\}$ is *conflicting* at claim p at time t iff its members are descendants of p, they are active at time t, and they cannot all be honored in full without exceeding the resource delegated to p:

$$\sum_{i=0}^{n} c_i.rset.count > p.rset.count$$

A set of tickets is *conflicting* iff their final claims $\{f_0, ..., f_n\}$

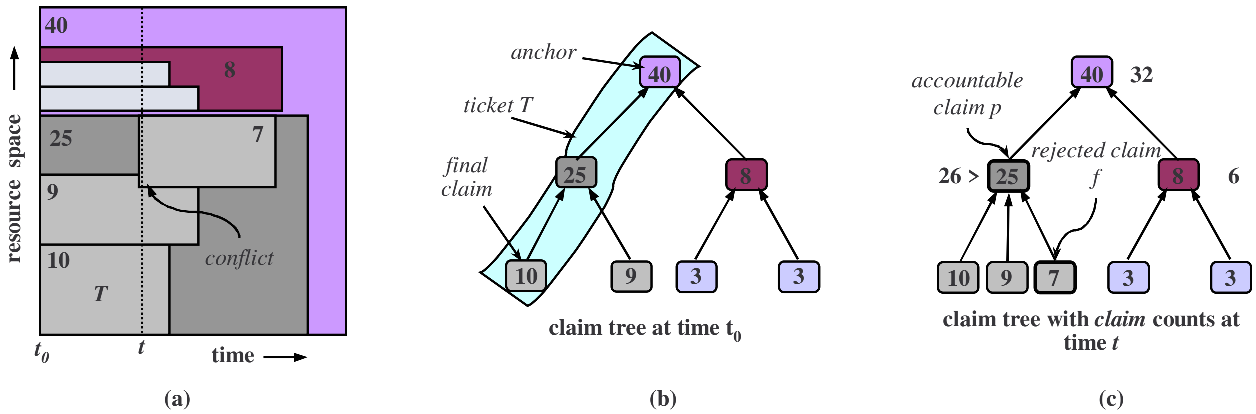

Figure 5: Example of nested claims. Part (a) depicts claims as consuming a share of the resource space for some period of time, with each subclaim contained within its parent claim. Parts (b) and (c) show the corresponding claim trees for time t_0 and t, with the conflict detected at time t.

are conflicting for some common ancestor p at some time t. Note that t and p are not necessarily unique for a given set of conflicting tickets. We consider the nearest (youngest) common ancestor p at the earliest conflict time t to be *accountable* for the conflict.

Given any set of conflicting tickets, it can be shown that the holder of the accountable claim p has oversubscribed its resources at time t, making the conflict possible. It follows from the definition of the accountable claim and the containment property of subclaims that p is itself oversubscribed at time t: p's immediate children are conflicting at p at time t. More strongly, the subset of p's children from which the tickets descend are conflicting at p at time t.

Figure 5 and Table 6 illustrate a simple algorithm to check for conflicts across any claim set. It maintains a *claim count* field in each cached claim record: the tickets descending from a claim p collectively commit *p.claim* units of resource at time t. When adding a ticket's final claim to the claim tree, it propagates the newly claimed resource count up through the tree, checking for conflicts. The state required is linear in the total number of tracked claims. The cost to validate a ticket and check it for conflicts is linear with the length of the ticket's chain. Agent implementations may use the claim tree structure in a similar way to track how they have delegated their resource holdings to their clients.

The conflict detection algorithm generalizes to claims whose terms start in the future. *Future claims* can support advance reservations and a futures market for a resource economy. Conflict detection for future claims works by rolling the claim tree forward or backward in time, adding claims that become active and removing claims that expire. The algorithm maintains an *interval list* that can return an ordered sequence of *start* and *end* events for the final (leaf) claims over any time interval. To determine if a ticket and its final claim f conflict with the existing ticket set, roll the claim tree forward to f's start time, then roll forward to its end time, checking for conflicts as each claim becomes active. The cost to check a set of claims for conflicts over any interval is proportional to the depth of the claim tree times the number of claims starting or ending during the interval. Many optimizations are possible to reduce the number of events to process, e.g., maintain separate interval lists for distinct subtrees of the claim tree.

```
p = f.parent;
while (p) {
   p.claim += f.rset.count;
   if (p.claim > p.rset.count) {
       reject f
       p is accountable
   }
   p = p.parent;
}
```

Table 6: Check a ticket's final claim f for conflicts with known claims in the claim tree.

If a conflict exists, the set of claims in the subtree rooted in the accountable claim p constitutes proof that a ticket was properly rejected and that the holder of p was the immediate cause of the conflict. Any entity can check the proof by scanning the claim set and building the claim subtree. The cost scales with the number of claims times the depth of the subtree.

3.5 Security Analysis and Discussion

Table 7 summarizes how the security mechanisms (Table 4) defend against each element of the threat model (Table 3) given the system's assumptions (Table 2). While most of the entries in Table 7 are easily checked by inspection, we note several fine points. A3 guarantees the integrity of SHARP protocol messages passing between trust domains, assuming that private keys are not compromised. In addition, the requirement to substantiate a ticket rejection ensures non-repudiation by a malicious site authority that might otherwise simply reject and omit evidence supporting the rejection (T8). Failure to supply evidence for ticket rejection can be taken as evidence of misbehavior by the site authority.

The non-repudiation (NR) properties are essential to detecting misbehavior and proving it to third parties so that economic and social mechanisms can control the threats, as discussed in Section 2.4. For example, since tickets are self-describing, any actor can verify that a principal issued a ticket. One complicating factor for non-repudiation in self-certifying systems is that an attacker can attempt to create multiple public keys to hide its identity; this is known as a sybil attack [19]. Sybil attacks are possible when there is

Threat		AUTH	IN	NR
T1A	Unauthorized local use	A1; A4; A5; M1; M2; M3; M6	M1; M2; M3; M6	
T1B	Unauthorized remote use	A1; A3; A4; A5; M1; M2; M3; M4; M6	M1; M2; M3; M4; M6	
T2	Man-in-the-middle attack	A1; A4	M1; M2; M3; M4; M6	
T3	Unauthorized agent use	A1; A2; A3; A4; M1; M2; M6	-	A1; A4; M6
T4	Malicious or untrusted site	A1; A2; A3; M3; M4	-	A7
T5	Malicious agent	M3; M4; M6	M3; M4; M6	A1; A3; A4
T6	Malicious service manager	M7	M7	A1; A3; A4
T7	Site authority inflates tickets	-	-	A1; M5; M6; M7
T8	Site authority rejects valid tickets	-	-	A1; M1; M2; M7
T9	Denial of service attack	A6	-	

Table 7: For each threat or attack enumerated in the threat model, this table identifies the assumptions and mechanisms that jointly ensure that Sharp provides the required authorization (AUTH), integrity (IN), or non-repudiation (NR) properties.

no centrally trusted authority to bind each public key to a fixed identity subject to external control. Sharp assumes no such central authority, but it is resistant to sybil attacks to the extent that actors authenticate their peering partners. In general, the fixed identity bound to a public key is known to the agent that certified the key, except in the case of anonymous economic transactions.

Sharp currently does not support revocation of keys or certificates. If a key or certificate is revoked, then any claims issued to it will continue to be valid unless the site authorities are notified. To prevent this, any agents that granted tickets to a revoked principal could notify the site authorities for those tickets, so that they can reject or revoke the claims. Each authority must store the revoked principal in a Certificate Revocation List (CRL) until all claims held by the revoked principal have expired.

The transitive trust relationships resulting from ticket delegation are subject to the *confinement problem* of classical capability systems [31]. That is, the owner of the resource (the site authority) cannot prevent an agent from carelessly certifying some malicious entity to access the resource. This is of particular concern for anonymous actors. As previously stated, Sharp limits this trust to control over physical resources for a bounded time. In particular, a Sharp ticket does not confer access to any shared object or shared state, so a malicious ticket holder cannot affect any other user if the site schedulers properly isolate guest services running in different slices. Of course, the ticket holder may attempt to abuse the resources, e.g., to mount a denial-of-service attack on a third party (T9B). The site authority is responsible for isolating slices and monitoring resources to prevent abuse (A6).

Although confinement is less important in Sharp than in general capability systems, it may be useful for an agent to further weaken the trust placed in a delegate. One way to achieve this is to extend the claim record to include more restrictions according to the issuer's policies. For example, an agent could mark a claim to notify downstream delegates that a certified principal is anonymous, or to explicitly limit its right to subdivide the ticket or certify other principals. Such certified security assertions are common in systems for general decentralized trust management [10] (see Section 6).

The accountability mechanisms in Sharp do not extend to correct functioning of the allocated resources themselves. In particular, Sharp assumes that clocks are synchronized within some tolerance that is small relative to the granularity of claim terms. For example, a site with a fast clock may expire its claims prematurely, or a service manager with a slow clock may falsely believe that the site shortchanged it. Some external mechanism must exist to monitor resource providers (sites) for compliance with their contracts (A7). This monitoring is essential in any service economy.

4. RESOURCE AVAILABILITY AND EFFICIENCY

This section discusses Sharp's mechanisms to achieve both high availability and high efficiency of target resources. Sharp claims may be viewed as *soft* reservations. Some form of reservation and/or admission control is needed to meet the goal of predictable performance. Unfortunately, hard reservations can compromise the efficiency and availability of resources. Resource availability suffers if a reservation manager or agent fails, preventing users from reserving the resources it controls. Resource efficiency suffers when the holder of a reservation fails or loses the reservation, denying the resources to other users. We rely on economic and social incentives to address the related problem of principals who claim excess resources and leave their leased resources underutilized (see Section 2.4).

Two key techniques reduce these dangers of reservations. First, Sharp uses soft-state claims of fixed duration, following the classical leases model [25]. Timed claims improve resource availability because any claims held by a failed entity will eventually expire, releasing its resource holdings for use elsewhere in the system. Claim expiration is also an opportunity for agents, site authorities, and service managers to renegotiate the resource contract. Thus timed claims are a basis for dynamic resource management that adapts to changing load and system conditions. In previous work we explore approaches to adaptive feedback-controlled resource managers that periodically grow or shrink resource slices according to load on the guest service [13, 20]; these systems could run as Sharp agents. The choice of claim duration is a tradeoff between agility, robustness, and renewal overhead. Shorter terms allow principals to renegotiate contracts more frequently, and they reduce the time to recover a resource if a claim holder fails, but they increase the overhead for resource management. Previous work on leases explores these tradeoffs [25, 51].

The second technique of *oversubscription* allows site authorities and agents to issue tickets that exceed their resource holdings. We define the *oversubscription degree* OD for a claim p at time t as the ratio of the aggregate number of resource units claimed for its issued children to the total number of resource units in the parent claim p. For example, an agent holding 100 units of resources may grant tickets for all 100 units to two different peering partners with the same terms, for an $OD = 2$ or $OD = 200\%$. Those partners may in turn subdivide the claims. If enough of the oversubscribed tickets are lost or allowed to expire without being redeemed, then the authority will honor all remaining claims. Thus oversubscribed tickets confer access to resources with configurable degrees of assurance. They bridge the continuum between reservations and "hints" from a status monitoring service. Oversubscription is related to *overbooking* [41], a distinct and complementary technique to improve resource efficiency when guest services do not consume all of the resources leased to their slices (see Section 6).

Oversubscription can improve the efficiency of distributed resource discovery. It allows a site or agent to advertise its resources aggressively by "pushing" claims for blocks of resources to multiple peer agents, making it easier for clients to discover them. Aggressive advertising of oversubscribed tickets can decrease the latency and overhead of resource discovery by reducing the search distance to locate a ticket for a suitable resource. Varying OD balances aggressive advertising with the assurance that a ticket will be honored once it is found.

Oversubscription can also improve resource availability. For example, replicating tickets at multiple agents improves the likelihood that a service manager may obtain the resources even if an agent fails. Again, this replication increases the risk that a site authority will reject a ticket when the holder attempts to redeem it for a lease. The experiments in Section 5 illustrate and quantify the effect of oversubscribing tickets on resource availability and efficiency in our prototype for resource bartering in PlanetLab.

Peer agents holding oversubscribed tickets may notify one another of their issued tickets to reduce the likelihood of granting conflicting tickets. For example, peer agents holding a replicated ticket set may coordinate to maintain consistency within some configurable bound on OD to balance coordination overhead with the risk of ticket rejection. A lower OD bound reduces the risk of ticket rejection but requires tighter consistency with higher coordination overhead. $OD = 1$ requires synchronous coordination; $OD >> 1$ reduces coordination overhead at the price of a higher risk of rejection. In this case there is a close relationship between OD and consistency metrics for bounded weak consistency [52]. For example, a replica set may bound OD for the tickets it issues by propagating updates as often as necessary to stay within some *numerical error* threshhold specified for each agent. A replica's numerical error limits the maximum number of tickets (or resource units encoded within the tickets) that a peer replica may issue without informing the local replica. Tighter numerical threshholds impose higher coordination overhead, and the overhead to maintain any given threshhold grows with the ticket request rate. Alternatively, the replica set can propagate updates to stay within a *staleness* bound for each replica, which bounds the delay for updates from its peers. In this case, the overhead is less sensitive to the ticket request rate, but higher request rates may increase OD. Previous work has quantified tradeoffs between coordination overhead and consistency in similar settings. We leave a study of oversubscription policies for replicated ticket sets to future work.

5. CASE STUDY: PLANETLAB

We have implemented a complete instance of the SHARP framework to perform resource discovery and authenticated resource allocation in the PlanetLab [36] wide-area testbed. The primary goals of our PlanetLab implementation are to: i) provide secure transparent access to global PlanetLab resources by all authorized principals, ii) lower the barrier to entry to joining the infrastructure, iii) maintain local site autonomy, and iv) enable a broad range of resource allocation policies. Our infrastructure and experiments are designed to demonstrate the generality and efficiency of SHARP. It is also important to note that SHARP generalizes beyond our PlanetLab implementation; the results presented in this section are for one implementation and for one of many possible policies enabled by the SHARP framework. We use this prototype simply to demonstrate and to evaluate the key properties of SHARP.

As of July 2003, PlanetLab consists of 160 machines spread across 65 world-wide locations. All PlanetLab machines run linux 2.4.19. Individual machines are the ultimate arbiters of resources in PlanetLab, with *node managers* running on each PlanetLab machine acting as SHARP site authorities. When a service manager successfully redeems a ticket for a lease, the node manager creates a virtual machine for the slice on the local node, with CPU, memory, and network resources according to the resource set described in the lease. The node manager also creates an account on the virtual machine matching a name in the request, and enables ssh access using the public key of the lease holder. We currently use vservers [42] to create lightweight virtual machines, though virtual machine technology is an active area of research [21, 44, 47]. A SILK [7] linux module assigns the specified resources to each vserver.

We implemented our SHARP prototype for PlanetLab in the Python language. Our implementation uses simple Python database objects to store active tickets at each agent and node manager. While a different implementation platform would likely improve the baseline performance of our prototype, our measurements below indicate that this implementation is sufficient for the system's target use in the PlanetLab environment with resource claims at the granularity of minutes or hours.

5.1 Transparent Access to Global Resources

PlanetLab node managers grant tickets for their resources to one or more peering agents. In typical use we expect one logical (possibly replicated) agent per autonomous system. Agents collect tickets from node managers and advertise a configurable portion of their ticket holdings globally, holding the remainder in reserve for clients in the same AS. Clients request slices through their service managers, which requests resources at specified sites in XML requests to the local agent. Each agent authenticates the requesting principal and grants tickets, which the service manager in turn redeems for leases with their node managers.

This simple scheme does not permit a service manager to obtain resources that are not held by its local agent. We extend the scheme with a simple resource routing mechanism based on pair-wise peering relationships among agents, negotiated in the same manner as ISPs exchange network bandwidth. For instance, an agent A might grant 10 units of A's resources to agent B in exchange for access to 10 units of B's resources. We encode the terms of this peering relationship in two SHARP tickets. Denote a ticket from a node manager local to B (NM_B) to agent B as $\{NM_B \rightarrow Agent_B\}$. Then B can create a chained ticket transferring a subset of the resources to an agent at site A: $\{\{NM_B \rightarrow Agent_B\}\{Agent_B \rightarrow Agent_A\}\}$. This pair-wise resource bartering enables $Agent_A$ to distribute claims for resources at B to its local users. More importantly, it enables any principal to access any global resource that is reachable through some transitive peering path, reminiscent of how Internet routing protocols enable universal packet delivery among tens of thousands of ASes that typically are not directly connected.

5.2 Secure Resource Routing

To enable such request routing in PlanetLab, we have implemented SRRP, the Secure Resource Routing Protocol. Our PlanetLab agents speak SRRP with remote peering partners to build simple routing tables for request routing. The goal of SRRP is to determine the next hop toward the agent holding tickets for a desired set of resources.

While the implementation of SRRP is involved, it is a straightforward adaptation of link state IP routing protocols such as OSPF. At a high level, agents monitor the link state of all edges incident to them and advertise this information to their peers. The link state advertisement includes the link connectivity and the amount of exchanged tickets through the link, summarizing the exchange rate and availability of tickets via this link. When a new agent joins an agent network, it first establishes a link with a pre-existing agent and then synchronizes its link state database with the agent. The existing agent propagates this new link state to all other agents in the network. Remote agents correspondingly adjust their routing table for this new update. An SRRP routing table can be built based on multiple metrics for an end-to-end path, including: i) the end-to-end exchange rate for remote tickets incorporating the exchange rate of individual intermediaries along the path, ii) the tickets available on an end-to-end path, or iii) the number of hops of a path to the target resources.

SRRP uses a customizable edge weight function that combines all metrics to compare paths for the same destination when building a routing table. SRRP also maintains multiple paths to each site, allowing it to consider alternate routes in case of failure or the exhaustion of tickets along a particular path. With global link state knowledge, each agent can individually determine a path to a site according to its local view of global state and its preferences. Thus, SRRP supports source-based request routing. Finally, SRRP supports next-hop forwarding in a best-effort manner, allowing sources to specify a set of constraints on cost or other metrics.

In contrast to traditional link state protocols, SRRP incorporates security as a first-class feature, with the goal of preventing malicious or compromised agents from subverting the routing mechanism. When an agent issues a link state update for an edge incident to it, it must sign the update message with its private key. Other agents can verify the integrity of the message. Thus, a malicious agent cannot forge or modify link states for edges starting from other agents. On the other hand, if a malicious agent advertises false link states about its relationships with other agents, the affected peers will eventually detect the attack (link state updates are eventually distributed throughout the system) and notify other agents, optionally through the aid of a reputation system.

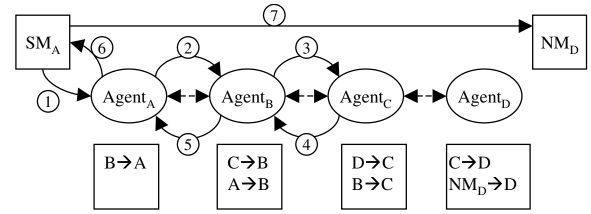

Figure 6: A sample PlanetLab deployment scenario demonstrating access to global resources through pair-wise peering relationships.

5.3 Evaluation

5.3.1 Microbenchmarks

We now present an example scenario and evaluation of the basic SHARP mechanisms. Consider the following scenario summarized in Figure 6. A principal at site A operates through its service manager SM_A and wishes to gain access to resources at site D (e.g., on resources administered by a node manager NM_D). In this example, agents at sites A, B, C, and D have established three pair-wise peering relationships, indicated by dashed lines in Figure 6. Boxes below each agent in Figure 6 represent the set of tickets held by each agent as a result of its pre-existing peering relationships.

SM_A transmits a description of its resource requirements (at D) to its agent, $Agent_A$. Through SRRP, $Agent_A$ knows that it can obtain resources at $Agent_D$ through its peering partner $Agent_B$. For simplicity, we will assume that all resource units are comparable (normalized to some baseline) and that peering exchanges are done on a one-for-one basis. Given this setup, $Agent_A$ transmits a request to $Agent_B$ describing its desired resources at $Agent_D$ along with a ticket describing an equal set of rights for resources at $Agent_B$. In this example, $Agent_B$ does not possess any tickets for resources at $Agent_D$, but through SRRP it knows that it can obtain $Agent_D$'s resources through its peering relationship with $Agent_C$. Thus, it marks the tickets transmitted by $Agent_A$ as "spent" (by updating local stable storage) and sends a recursive request for resources at $Agent_D$ to $Agent_C$, along with a ticket describing $Agent_B$'s resource privileges at $Agent_C$. $Agent_C$ has direct access to resources at $Agent_D$ through its direct peering relationship. It honors its peering relationship with $Agent_B$ and creates a new ticket granting $Agent_B$ access to $Agent_D$'s resources. In step 6, $Agent_B$ creates another ticket transferring the privileges back to $Agent_A$, who in turn creates a ticket granting the appropriate access to SM_A. Finally, SM_A directly contacts NM_D and redeems the ticket for a lease. As part of this process, NM_D creates a virtual machine and a login for SM_A.

We ran the above scenario across four PlanetLab nodes, each in a separate administrative domain, in the United States. SM_A and $Agent_A$ were co-located on one machine while NM_D and $Agent_D$ were co-located on another. $Agent_B$

Step	Node	Operation	Cost
1	A	Local search	.1ms
1	A	Create barter ticket	98.6 ms
2	B	Mark ticket spent	62.0 ms
3	C	Local search and delegate	101.7 ms
5	A	Validate ticket	37.7 ms
2-5	A	Receive ticket	752.7 ms
6	A	Delegate to SM_A	178.0 ms
7	NM_D	Validate ticket	56.8 ms
7	NM_D	Check conflict in tree	91.7 ms
7	NM_D	Create lease	45.5 ms
7	NM_D	Insert ticket into tree	2.8 ms
7	NM_D	Create vserver	2 sec

Table 8: A breakdown of various operations involved in the scenario depicted in Figure 6.

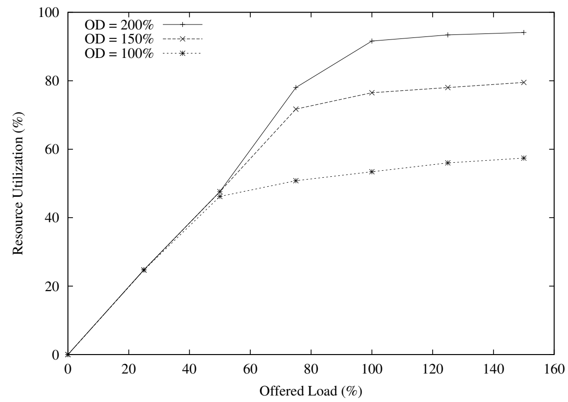

Figure 7: Resource efficiency as a function of offered load and oversubscription degree for a sample PlanetLab scenario.

and $Agent_C$ ran on separate machines. Table 8 shows the cost breakdown for selected steps in this scenario. The end-to-end cost (steps 1-6 of Figure 6) for SM_A to receive a delegated ticket for resources at NM_D is 1.15 seconds. In step 1, $Agent_A$ searches its local database for tickets from D. This search fails, taking 0.1 ms. At this point $Agent_A$ consults its routing table and creates an XML representation of a ticket (for resources at $Agent_B$) and transmits a request to $Agent_B$, taking 98.6 ms. $Agent_B$ then begins the recursive process of obtaining the appropriate ticket from its own peer, taking a total of 752.7 ms in steps 2-5. When the requested ticket is sent back by $Agent_B$, $Agent_A$ spends 37.7 ms (step 5) to verify the ticket. Then it updates its own database of issued tickets and creates a delegated ticket for transfer to SM_A, which takes 178.0 ms (step 6). Python's XML parsing and database implementation form the major overhead components for most of these operations. For example, within the 37.3 ms overhead for validating the requested ticket, only 2.0 ms was spent in verifying the ticket's content and signature. The remaining overhead comes entirely for parsing the XML format of the ticket.

The total end-to-end latency incurred by SM_A to obtain a lease from NM_D in step 7 was 0.63 seconds in this experiment. The time for NM_D to create a local virtual machine took 2 seconds in our experiment. Vservers create a set of "copy on write" links to a main file system (using `chroot` to set the root file system for the vserver). Typically, initializing a vserver from scratch takes between 15-60 seconds, depending on the complexity of the virtual machine. For our experiments, we have implemented a simple technique for "pre-loading" common vserver configurations, allowing us to hide most of the latency.

5.3.2 Oversubscription

The next experiment explores the effect of oversubscribing tickets on availability and efficiency (see Section 4). This experiment uses a connected mesh of 41 PlanetLab nodes, spread across the United States. A randomly chosen agent S acts as the "sink" for resource requests originating from service managers co-located with 10 other randomly chosen agents. Each agent other than the sink randomly peers with between 4 and 7 remote agents. We assume that S has a capacity of 50 resource units and that it peers with 2 randomly chosen agents. S evenly delegates tickets representing all its resource privileges (saving none for local use in this example) among its two peers according to some oversubscription degree, OD. With $OD = 100\%$, S delegates 25 resource units to each of its peers, in effect advertising 100% of its resources. With $OD = 200\%$, S delegates 50 resources units to each peer.

The 10 service managers wishing to consume resources at S generate requests according to a Poisson arrival rate. The remaining 30 agents simply act as forwarders of requests based on SRRP, bartering tickets to reach one of S's peers (as in the previous example). The *uniform* study evenly distributes requests from all service managers. In the *skewed* study, 3 service managers generate 90% of the requests. Each request obtains one unit of resource for a fixed term of 200 seconds. Each experiment runs for 20 minutes.

Figure 7 plots resource utilization at S as a function of offered load for the skewed request load. We vary offered load such that the sum of the means of the request arrival rate across the 10 service managers results, on average, in some selected percentage of S's capacity. For this configuration, the three service managers generating 90% of the requests all happen to have a path to the same peer of S. Thus, for $OD = 100\%$ where each peer only has 25 resource units, one peer receives requests near its capacity while another peer is largely idle. Figure 7 shows that as we increase offered load from 50% to 150% of S's capacity, utilization only improves from 46.2% to 57.4%. As we increase OD from 100% to 200%, resource efficiency increases with the offered load. We omitted the graph for the uniform load, whose utilization increases linearly with offered load as expected, largely independent of OD.

Figure 8 shows the breakdown of four possible outcomes for a service manager request. *Success* indicates that both a ticket and a lease were successfully obtained. There are three failure conditions. *Oversubscribe* indicates that a ticket was successfully obtained from a remote agent, but the node manager rejected it as a conflict. A *True Negative* indicates that an agent rejected a ticket request for a fully utilized resource while a *False Negative* indicates that an agent rejected a request for a resource that was unclaimed.

As offered load increases, the ratio of successes to failures decreases independent of OD. Since resources are limited, higher load increases the ratio of rejected requests. All

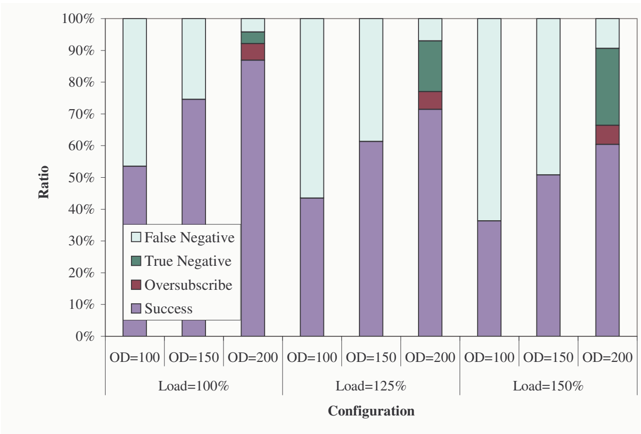

Figure 8: Breakdown of the result of resource requests for a sample oversubscription scenario.

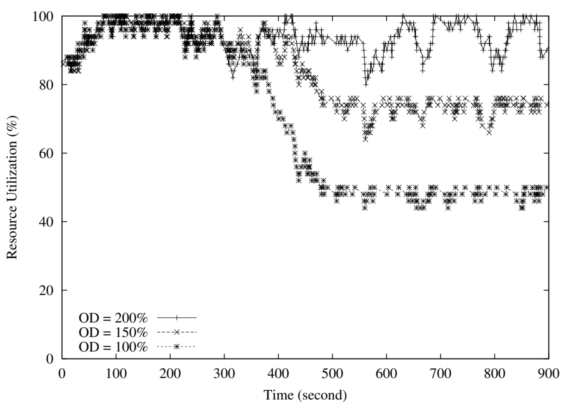

Figure 9: Oversubscription compensates for agent failure.

failures for $OD = 100\%$ and $OD = 150\%$ are false negative because the workload implies that resources are still available at S. For this experiment, when oversubscription reaches $OD = 200\%$, S begins to reject conflicting tickets. In this example, oversubscription improves resource efficiency. However, had some agent maliciously oversubscribed tickets, S would detect the conflict and reject only those requests delegated through the misbehaving agent.

5.3.3 Failure

Finally, we consider the effect of oversubscription on resource availability in the case of failure. We run the same experiment as above for the uniform workload case with an offered load 125% of the capacity of S, and manually kill one agent at time $t = 300$ (after a warm up period). Figure 9 plots resource utilization at S as a function of time for three values of OD. We see that for $OD = 100\%$, the failure causes resource utilization to fall to 50% as existing jobs drain from the system; 50% of resources become unavailable when the agent holding tickets for those resources fails. As we increase OD, resource utilization recovers to 75% for $OD = 150\%$ and near 100% for $OD = 200\%$ as SRRP discovers the alternate agent holding oversubscribed replicas of tickets held by the failed agent. Ideally, the system would adjust OD to maintain a target level of efficiency (e.g., accounting for possible failures), with acceptably low conflict rates.

6. RELATED WORK

SHARP defines a comprehensive architecture for managing distributed resources, combining cryptographically protected authorization, leases, hierarchical resource management, probabilistic resource tickets, and self-certifying decentralized trust management. It is related to many previous systems in several areas.

Reservations and shares. SHARP assumes resource control and scheduling mechanisms at each site to enforce the resource shares granted to claims and slices. Many systems have defined suitable abstractions for resource slices on individual nodes [6, 21, 7, 44, 30] or across groups of nodes [4, 11, 36, 5, 14]. Several works also define proportional share scheduling mechanisms to enforce resource assignments across slices; a classic example is lottery scheduling [46]. SHARP is complementary: it is a framework for assigning resources to slices according to decentralized policies for resource management and access control.

Leases. SHARP claims are based on the *lease* [25, 51] mechanism for cache consistency. SHARP adopts leases for soft-state ownership of distributed resources, like Jini [43] and recent grid proposals [23, 18, 22, 8]. SHARP extends these systems by cryptographically signing claims to make them unforgeable, non-repudiable, and independently verifiable by third parties. SHARP claims are distributed and hierarchical; the idea of distributed hierarchical leases was proposed by Lampson [28], and they are also used for wide-area cache consistency [51].

Capabilities. SHARP claims are signed assertions of authority that may be delegated to third parties. They are similar to capabilities [31] with key differences as outlined in Section 2.1. Cryptographic distributed capabilities first appeared in the Amoeba [40] distributed operating system, and other extensible operating systems use capabilities as well (e.g., [2]).

Cryptographic tickets. Kerberos [39] was an early system to introduce cryptographic tickets to authenticate a client for access to a server. SHARP tickets are similar but they assert limited privileges for the holder with respect to the server. Many systems use signed assertions to specify access privileges, including CRISIS [9], NASD [24], the Globus Community Authorization Service (CAS) [35], and systems for decentralized trust management discussed below. SHARP applies this concept to resource ownership and control.

Hierarchical resource management. SHARP resource managers (agents) may subdivide their resource claims and delegate them to other parties in a hierarchical fashion. Exokernel [26] uses hierarchical capabilities to support fine-grained control over resource access. Lottery scheduling [46] defines a notion of resource tickets that may be subdivided hierarchically; SHARP tickets are related but distinct in fundamental ways (see below). Systems based on these ideas demonstrated that hierarchical delegation is a key enabler for flexible resource management, but with few exceptions (e.g., [53]) they are limited to a single node; SHARP extends these ideas to wide-area resource management.

Probabilistic tickets. SHARP agents may issue arbitrary numbers of tickets against their resource holdings. Resource managers in lottery scheduling [46] use a similar technique for flexible ticket delegation. A key difference is that issuing more lottery tickets causes inflation; all tickets in a domain have the same value, but they all lose value. An oversubscribed SHARP ticket retains its face value with some probability that the site authority will reject it entirely. In this case the authority notifies the ticket holder of the rejection; the victim may request compensation for the rejected ticket or obtain its resources elsewhere rather than suffering unexpectedly degraded service. SHARP tickets differ in other ways: they can encode any number of resource units and they support advance reservations in that the holder may store a ticket and redeem it in the future (until it expires).

Overbooking. Oversubscribing SHARP tickets improves resource efficiency when some allocated resources are left unclaimed. This concept is related to *overbooking*, in which the scheduler at a site commits the same resources to multiple slices. Previous work [41] shows that significant efficiency gains are available with relatively modest conflict rates for some workloads. SHARP accommodates overbooking: overbooking applies to leases at the discretion of the site authority, while oversubscription applies to tickets. Overbooking degrades service when a conflict occurs, while conflicts on oversubscribed tickets typically result in explicit rejection.

Computational economies. While the literature on economic resource allocation is too large to survey here, research systems using economic models include Amoeba [40], Spawn [45], Muse [13], Millennium [15], and many systems based on lottery tickets and currencies [46]. A key goal of SHARP is to create a foundation for computational economies in wide-area systems without global trust [49]. SHARP is powerful enough to support a rudimentary barter economy, but a true computational economy requires additional payment mechanisms (e.g., digital cash).

Resource discovery and the Grid. SHARP and SRRP are closely related to frameworks for resource discovery [3, 17, 22, 43]. Like intentional naming (INS [3]), SRRP approaches resource discovery as a routing problem similar to BGP. Current Grid computing systems base resource management on resource discovery, e.g., as implemented in the Globus Monitoring and Discovery Service or MDS [17]. Like MDS, SHARP proposes a network of brokers, but it supplements the hints provided by MDS with support for cryptographically secure soft-state reservations.

Differentiated service. SHARP claims allow resource managers to provision wide-area networked resources to meet end-to-end service quality targets (SLAs). Most of the large body of research on differentiated service is applicable within individual SHARP sites or at the network level. Interest is growing to apply these mechanisms to wide-area computing resources for on-demand utility computing [5, 13, 20, 11, 49] or the Grid [18, 22, 23]. In particular, SHARP shares the key goals of the Globus SNAP proposal [18]. SNAP proposes to meet end-to-end SLA targets by negotiating sub-SLAs for an application's components with the sites running them; SHARP allows a resource manager to assemble the claims on global resources needed to meet end-to-end service quality targets directly.

Self-certifying trust delegation. SHARP is also related to PolicyMaker [10] and other pioneering decentralized trust management systems that integrate authorization with chained certificate delegation [27, 33, 12, 9, 50]. The basic mechanism in these systems is a delegated *trust certificate*, which certifies the public key for the new principal (a delegate or proxy) and defines its authorized actions, which may reflect weakened trust. SHARP tickets extend this idea to secure distributed resource management by allowing an agent to certify another principal to control a limited subset of its resource rights. Note that claims could be extended to incorporate other restrictions as in PolicyMaker, e.g., to restrict a principal's right to delegate its claims or certify other principals. Finally, SHARP builds on the idea of a distributed web of trust with no central authority [32, 54].

Peer-to-Peer Resource Allocation. Our work also relates to resource allocation in peer-to-peer systems, which require incentives for participants to contribute resources. Ngan et al. [34] suggest cooperative audits to ensure that participants contribute storage commensurate with their usage. Samsara [16] considers storage allocation in a peer-to-peer storage system, and introduces cryptographically signed storage claims to ensure that any user of remote storage must devote a like amount of storage locally. While both techniques center around audits, it is not yet clear how to apply these techniques to *renewable* resources such as CPU and bandwidth, as in SHARP. In a position paper, Shneidman et al. [38] propose economic incentives and mechanism design to encourage participants to contribute resources. Relative to all these efforts, SHARP operates under a different set of assumptions. SHARP operates at the granularity of autonomous systems or sites, which are reasonably long-lived. To join the system a SHARP site must negotiate resource contracts with one or more existing group members. These contracts, in effect, specify the system's expectations of the site and the site's promise of available resources to the system. Accountable claims make it possible to monitor each participant's compliance with its contracts, simplifying audits and making collusion more difficult in SHARP relative to general peer-to-peer systems.

7. CONCLUSION

This paper presents the design and implementation of SHARP, an architecture for Secure Highly Available Resource Peering. We make three principal contributions. First, we define self-certifying secure claims to global resources that can be safely delegated and traded across trust domains. Properties of these claims enable flexible tradeoffs between resource utilization, availability, and conflict. The security mechanisms protect against a variety of attacks that may be mounted against individual actors in the system. Second, we show how to build a computational economy among individual trust domains through pair-wise peering arrangements and resource bartering. Finally, we demonstrate the practicality of our approach in large-scale experiments with a SHARP prototype running across the PlanetLab wide-area testbed.

Acknowledgments

This paper improved significantly from the detailed comments provided by our shepherd Larry Peterson and the anonymous reviewers. David Becker assisted us with experiments and data processing. Adriana Iamnitchi, Jim Horning, Justin Moore, and John Wilkes provided valuable comments on earlier drafts of this paper.

8. REFERENCES

[1] Karl Aberer and Zoran Despotovic. Managing Trust in a Peer to Peer Information System. In *Tenth International Conference on Information and Knowledge Management*, November 2001.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation For UNIX Development. In *Proceedings of the 1986 USENIX Summer Conference*, pages 93–112, June 1986.

[3] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The Design and Implementation of an Intentional Naming System. In *Symposium on Operating Systems Principles*, December 1999.

[4] D. Scott Alexander, Paul B. Menage, Angelos D. Keromytis, William A. Arbaugh, Kostas G. Anagnostakis, and Jonathan M. Smith. The Price of Safety in an Active Network. *Journal of Communications and Networks*, 3(1):4–18, March 2001.

[5] K. Appleby, S. Fakhouri, L. Fong, G. Goldszmidt, M. Kalantar, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - SLA Based Management of a Computing Utility. In *Proceedings of the 7th IFIP/IEEE International Symposium on Integrated Network Management*, May 2001.

[6] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implementation*, February 1999.

[7] Andy Bavier, Thiemo Voigt, Mike Wawrzoniak, Larry Peterson, and Per Gunningberg. SILK: Scout Paths in the Linux Kernel. Technical Report 2002-009, Department of Information Technology, Uppsala University, Uppsala, Sweden, February 2002.

[8] Micah Beck, Terry Moore, James S. Plank, and Martin Swany. *Active Middleware Services (Salim Hariri, Craig A. Lee, and Cauligi S. Raghavendra editors)*, chapter Logistical Networking: Sharing More Than the Wires. Kluwer Academic, Norwell, MA, 2000.

[9] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS Wide Area Security Architecture. In *Proceedings of the USENIX Security Symposium*, San Antonio, Texas, January 1998.

[10] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized Trust Management. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.

[11] Rebecca Braynard, Dejan Kostić, Adolfo Rodriguez, Jeffrey Chase, and Amin Vahdat. Opus: an Overlay Peer Utility Service. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH)*, June 2002.

[12] Randy Butler, Von Welch, Douglas Engert, Ian Foster, Steven Tuecke, John Volmer, and Carl Kesselman. A National-Scale Authentication Infrastructure. *IEEE Computer*, 33(12):60–66, December 2000.

[13] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing Energy and Server Resources in Hosting Centers. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 103–116, October 2001.

[14] Jeffrey S. Chase, Laura E. Grit, David E. Irwin, Justin D. Moore, and Sara E. Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the Twelfth International Symposium on High Performance Distributed Computing (HPDC-12)*, June 2003.

[15] Brent N. Chun and David E. Culler. User-centric Performance Analysis of Market-based Cluster Batch Schedulers. In *2nd IEEE International Symposium on Cluster Computing and the Grid*, May 2002.

[16] Landon Cox and Brian Noble. Samsara: Honor Among Thieves in Peer-to-Peer Storage. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[17] Karl Czajkowski, Steven Fitzgerald, Ian Foster, and Carl Kesselman. Grid Information Services for Distributed Resource Sharing. In *Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC)*, August 2001.

[18] Karl Czajkowski, Iand Foster, Carl Kesselman, Von Sander, and Steven Tuecke. SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, July 2002.

[19] John Douceur. The Sybil Attack. In *International Peer-to-Peer Symposium*, February 2002.

[20] Ronald P. Doyle, Omer Asad, Wei Jin, Jeffrey S. Chase, and Amin Vahdat. Model-Based Resource Provisioning in a Web Service Utility. In *Proceedings of the Fourth USENIX Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[21] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[22] Ian Foster, Carl Kesselman, Craig Lee, Robert Lindell, Klara Nahrstedt, and Alain Roy. A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation. In *Proceedings of the International Workshop on Quality of Service (IWQoS)*, June 1999.

[23] Ian Foster, Carl Kesselman, Jeffrey Nick, and Steven Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. In *Open Grid Service Infrastructure Working Group, Global Grid Forum*, June 2002.

[24] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[25] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 202–210, 1989.

[26] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, October 1997.

[27] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in Distributed Systems: Theory and Practice. In *The 13th ACM Symposium on Operating Systems Principles*, pages 165–182, October 1991.

[28] Butler W. Lampson. How to Build a Highly Available System Using Consensus. In Babaoglu and Marzullo, editors, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151, pages 1–17. Springer-Verlag, Berlin Germany, 1996.

[29] Seungjoon Lee, Rob Sherwood, and Bobby Bhattacharjee. Cooperative Peer Groups in NICE. In *IEEE INFOCOM*, April 2003.

[30] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996.

[31] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[32] David Mazières, Michael Kaminsky, M. Frans Kaashoek, and Emmett Witchel. Separating Key Management From File System Security. In *Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP)*, December 1999.

[33] B. C. Neuman. Proxy-Based Authorization and Accounting for Distributed Systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, May 1993.

[34] Tsuen-Wan Ngan, Dan Wallach, and Peter Druschel. Enforcing Fair Sharing of Peer-to-Peer Resources. In *Proceedings of the International Peer to Peer Symposium*, February 2003.

[35] Laura Pearlman, Von Welch, Ian Foster, Carl Kesselman, and Steven Tuecke. A Community Authorization Service for Group Collaboration. In *Proceedings of the IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, June 2002.

[36] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. In *Proceedings of ACM HotNets-I*, October 2002.

[37] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed Resource Management for High Throughput Computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 1998.

[38] Jeff Shneidman and David Parkes. Rationality and Self-Interest in Peer to Peer Networks. In *Proceedings of the International Peer to Peer Symposium*, February 2003.

[39] Jennifer G. Steiner, Clifford Neuman, and Jeffrey I. Schiller. Kerberos: An Authentication Service for Open Network Systems. In *Proceedings of the USENIX Technical Conference*, March 1988.

[40] Andrew Tanenbaum, Sape Mullender, and Robert van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, May 1986.

[41] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms . In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[42] Virtual Private Servers and Security Contexts. `http://www.solucorp.qc.ca/miscprj/s_context.hc`, 2002.

[43] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82, July 1999.

[44] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Symposium on Operating Systems Design and Implementation*, December 2002.

[45] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A Distributed Computational Economy. *Software Engineering*, 18(2):103–117, February 1992.

[46] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Operating Systems Design and Implementation*, pages 1–11, Novemeber 1994.

[47] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of Operating Systems Design and Implementation*, December 2002.

[48] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.

[49] John Wilkes, Patrick Goldsack, G. (John) Janakiraman, Lance Russell, Sharad Singhal, and Andrew Thomas. eOS: the Dawn of the Resource Economy. In *Workshop on Hot Topics in Operating Systems (HotOS)*, May 2001.

[50] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos Operating System. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 256–269, December 1993.

[51] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Arun Iyengar. Engineering Server-Driven Consistency for Large Scale Dynamic Web Services. In *Proceedings of the 2001 International World Wide Web Conference*, May 2001.

[52] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Conit-Based Continuous Consistency Model for Replicated Services. *Transactions on Computer Systems*, 20(3):239–282, August 2002.

[53] Tao Zhao and Vijay Karamcheti. Enforcing Resource Sharing Agreements among Distributed Server Clusters. In *Proceedings of the Sixteenth International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[54] Phil Zimmerman. PGP User's Guide. Online Documentation, 1994.