Co-Allocation, Fault Tolerance and Grid Computing

Jon MacLaren,[1] **Mark Mc Keown**,[2] and Stephen Pickles[2]

[1] *Center for Computation and Technology, Louisiana State University, Baton Rouge, Louisiana 70803, United States.*

[2] *Manchester Computing, The University of Manchester, Oxford Road, Manchester M13 9PL.*

Experience gained from the TeraGyroid and SPICE projects has shown that co-allocation and fault tolerance are important requirements for Grid computing. Co-allocation is necessary for distributed applications that require multiple resources that must be reserved before use. Fault tolerance is important because a computational Grid will always have faulty components and some of those faults may be Byzantine. We present HARC, Highly-Available Robust Co-allocator, an approach to building fault tolerant co-allocation services. HARC is designed according to the REST architectural style and uses the Paxos and Paxos Commit algorithms to provide fault tolerance. HARC/1, an implementation of HARC using HTTP and XML, has been demonstrated at SuperComputing 2005 and iGrid 2005.

I. INTRODUCTION

In this paper we discuss the importance of co-allocation to Grid computing. We provide some background on the difficulties associated with co-allocation before presenting HARC, Highly-Available Robust Co-allocator, a fault tolerant approach to co-allocation. The paper also includes a discussion on the problem of fault tolerance and Grid computing. We make the case that a computational Grid will always have faulty components and that some of those faults will be Byzantine [26, 27]. However, we also demonstrate that with suitable approaches it is still possible to make a computational Grid a fault tolerant system.

There are many definitions of Grid computing but recurring themes are: large scale or internet scale distributed computing and sharing resources across multiple administrative domains. The goal of sharing resources between organizations dates back to the ARPANET [33] project and progress towards that goal can be seen in the development of the Internet, the World Wide Web [22] and now computational Grids. While the goals of the ARPANET project are still relevant today the underlying infrastructure has changed, powerful computers have become cheap and plentiful while high performance networks have become pervasive, presenting developers with a different set of challenges and opportunities. We believe that co-allocation is a new challenge, while the falling cost of components makes fault tolerance a new opportunity.

Parallel to the evolution of ARPANET through to Grid computing has been the development of the theory of distributed systems providing us with a deeper understanding of distributed systems and a set of algorithms for building fault tolerant systems. Representational State Transfer [12], REST, is an architectural style for building large scale distributed systems that was used to develop the protocols that make up the World Wide Web. Paxos [28] is a fault tolerant consensus algorithm that can be used to build highly available systems [31]. Paxos Commit [18] is Paxos applied to the distributed transaction commit problem.

HARC uses REST and Paxos to provide a system that is fault tolerant and suitable for a large scale distributed system such as a computational Grid. HARC's focus on fault tolerance is unique among approaches to designing co-allocation services [3, 9, 24, 34, 39].

II. CO-ALLOCATION

Running distributed applications on a computational Grid often requires that the resources needed by the application are available at the same time. The resources may need to be booked (*eg* Access Grid nodes) or they may use a batch submission system (*eg* HPC systems). We define co-allocation as the provision of a set of resources at the same time or at some co-ordinated set of times. Co-allocation can be achieved by making a set of reservations for the required resources with the respective resource providers.

Experience from the award winning TeraGyroid [7] and SPICE [23] projects has shown that support for co-allocation on current production Grids [37, 38] is *ad-hoc* and often requires time consuming direct negotiation with the resource administrators. The resources required by TeraGyroid and SPICE included HPC systems, special high performance networks, high end visualization systems, Access Grid nodes, haptic devices and people. The lack of convenient co-allocation services prevents the routine use of the techniques pioneered by TeraGyroid and SPICE. Without co-allocation computational Grids are limited in the type of applications they can support, and so are limited in their potential. Co-allocation's im-

portance to Grid computing means that it must be a reliable service.

III. FAULT TOLERANCE AND GRID COMPUTING

Whatever definition of Grid computing is used we are lead to two inescapable conclusions: a computational Grid will always have faulty components and some of those faults will be Byzantine.

Computational Grids are internet scale distributed systems, implying large numbers of components and wide area networks. At this scale there will always be faulty components.

The case that a computational Grid will always have faulty components is illustrated by the fact that on average over ten percent of the daily operational monitoring tests, GITS [4], run on the UK National Grid Service, UK NGS [38], report failures. Since the start of the UK NGS a number of the core nodes have been unavailable for days at a time due to planned and unplanned maintenance.

Crossing administrative boundaries raises issues of trust: Can a user trust that a resource provider has configured and administrates the resource properly? Can a resource provider trust that a user will use the resource correctly? Distributed transactions are rarely used between organizations because of a lack of trust; one organization will not allow another organization to hold a lock on its internal databases. Without trust users, resource providers and middleware developers must be prepared for Byzantine fault behaviour: when a component faults but continues to operate in an unpredictable and potentially malicious way. Although Grid computing supports the creation of limited trust relationships between organizations through the concept of the *Virtual Organization* [14] our experience has been that Grids exhibit Byzantine fault behaviour.

To illustrate the point we provide three examples of Byzantine behaviour which we have encountered. The first case involved the UK eScience Grid's MDS2 [2, 10] hierarchy. A GRIS [10] at a site was firewalled preventing GIIS [10] higher up in the MDS2 hierarchy from querying it. The GRIS continued to report that it was publishing information but whenever a GIIS attempted to query it the firewall would block the connection. The GIIS would block waiting for the GRIS to respond causing the whole MDS2 hierarchy to block. The firewall was raised by the site's network administrators. The local firewall on the GRIS server and the GRIS service itself were configured correctly.

The second case involved a job submission node on the TeraGrid [37]. The node consisted of two servers and utilized a DNS round robin to allocate requests to a node. Unfortunately the DNS entries were mis-configured and reported the wrong hostname for one of the nodes causing job submissions to fail randomly. Local testing at the site did not reveal the problem.

The third case involved a resource broker that assigned jobs to computational resources. Occasionally a computational resource would fail in a Byzantine way: it would accept jobs from the resource broker, fail to execute the job but report to the resource broker that the job had completed successfully. The resource broker would continue assigning jobs to the faulty resource until it was drained of jobs.

The examples illustrate the importance of end-to-end arguments in system design — error recovery at the application level is absolutely necessary for a reliable system, and any other error detection or recovery is not logically necessary but is strictly for performance [35]. In each case making the individual components more reliable would not have prevented the problem. Retrofitting reliability to an existing design is very difficult [30].

For co-allocation a very real example of Byzantine fault behaviour occurs when a resource provider accepts a reservation for a resource but at the scheduled time the user, and possibly the resource provider, discover the resource is unavailable.

IV. REST

REST is an architectural style for building large scale distributed systems. It consists of a set of principles and design constraints which were used in designing the protocols that make up the World Wide Web.

We provide a brief description of REST and refer the reader to the thesis [12] for a complete description. REST is based on a client-server model which supports caching and where interactions between client and server are stateless, all interaction state is stored on the client for server scalability. The concept of a resource is central to REST, resources have identity and anything that can have an identity can be a resource. Resources are manipulated through their representations and are networked together through linking — hypermedia is the engine of application state. Together the last set of constraints combine to make up the principle of uniform interface. REST also has an optional constraint for the support of mobile code.

HTTP [11] is an example of a protocol that has been designed according to REST. On the World Wide Web a resource is identified by a URI [6] and clients can retrieve a representation of the resource using a HTTP GET. HTTP can also supply caching information along with the representation to allow

intermediaries to cache the representation. The representation may contain links to other resources creating a network of resources. Clients can change the representation of a resource by replacing the existing representation with a new one, for example using a HTTP PUT. All resources on the World Wide Web have a uniform interface allowing generic pieces of software such as web browsers to interact with them. Web servers are also able to send code, for example JavaScript, to the client to be executed in the browser.

V. PROBLEMS OF CO-ALLOCATION

To illustrate the problems associated with co-allocation we present two possible approaches and discuss their shortcomings. Most existing solutions [3, 9, 24, 34, 39] for co-allocation are based on variations of these approaches.

A. One Phase Approach

In this approach a successful reservation is made in a single step.

The user sends a booking request to each of the Resource Managers (RM) requesting a reservation. The RMs either accept or reject the booking. If one or more of the RMs rejects the booking the user must cancel any other bookings that have been made. This approach has the advantage of being simple but a potential drawback is that the user may be charged each time he cancels a reservation.

Supporting reservations prevents a RM from running the optimal workload on a resource as it has to schedule around the reservations. Even if a reservation is cancelled before the scheduled time it may already have delayed execution of some jobs. RMs may charge for the use of the resource and may charge more for jobs that are submitted via reservation to compensate for the loss in throughput. They may also charge for reservations that are cancelled.

Any charging policy is the prerogative of the RM. Not all RMs may charge and it is unreasonable to make any assumptions about or try to mandate a charging policy. A co-allocation protocol should accommodate the issues associated with charging but cannot depend on RMs supporting charging.

Another potential problem with this approach arises if one of the RMs rejects a booking but the user does not cancel the other reservations that have been made. For example the user may fail before he has had a chance to cancel the reservations. Even if the user recovers he may not have stored sufficient state before failing to cancel the reservations. This is related to the question of trust — the RMs must trust the user to cancel any unwanted reservations.

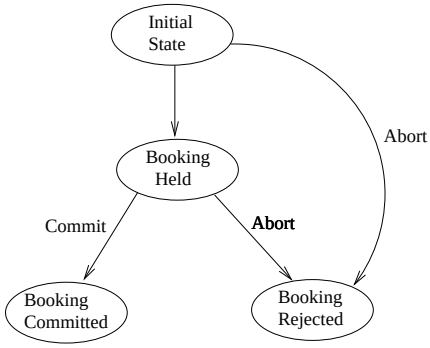

FIG. 1: The state–transition diagram for a RM in the two phase approach. The RM receives the booking request message in the initial state. It can either reject the request and move to the booking rejected state or accept the request and move to the *Booking Held* state. The RM waits in the *Booking Held* state until it receives the *Commit* or *Abort* message from the TM.

A partial solution is to add an intermediary service that is trusted by the user and RM to correctly make and cancel reservations as necessary. The user sends the set of bookings he requires to the intermediary which handles all the interactions with the RMs, cancelling all the reservations if the purposed schedule is unacceptable to any of the RMs. The intermediary is still a single point of failure unless it can be replicated.

B. Two Phase Approach

In this approach a successful reservation is made in two steps.

To resolve the problem of RMs charging for cancelling a reservation we introduce a new state for the RM, *Booking Held*. The client can cancel a reservation in the *Booking Held* state without being charged. The approach is similar to the two phase commit [17] algorithm for distributed transactions but does not necessarily have the ACID (Atomicity, Consistency, Isolation and Durability [19]) properties associated with two phase commit.

A process, the Transaction Manager (TM), tries to get a group of RMs to accept or reject a set of reservations. In the first phase the TM sends the booking requests to the RMs, the RMs reply with a *Booking Held* message if the request is acceptable or an *Abort* message if the request is unacceptable. If all the RMs reply with *Booking Held* the TM sends a *Commit* message to the RMs and the reservations are committed. Otherwise the TM sends an *Abort* message to the RMs and no reservations are made.

Fig. 1 shows the state transitions for the RM in the two phase approach.

Unfortunately the two phase approach can block. If the TM fails after the first phase but before starting the second phase, before sending the *Commit* or *Abort* message, the RMs are left in the *Booking Held* state until the TM recovers. While in the *Booking Held* state the RMs may be unable to accept reservations for the scheduled time slot and the resource scheduler may be unable to run the optimal work load on the resource.

The blocking nature of the two phase approach may be acceptable for certain scenarios depending on how long the TM is unavailable and providing it recovers correctly. To overcome the blocking nature of the two phase approach requires a three phase protocol [36] such as Paxos.

Another potential problem with the two phase approach is that RMs must support the *Booking Held* state. A solution is for RMs that do not support the *Booking Held* state to move straight to the *Booking Committed* state if the booking request is acceptable in the first phase. For the second phase they can ignore a *Commit* message and treat an *Abort* message as a cancellation of the reservation for which the user may be charged. This a relaxation of the consistency property of two phase commit.

It is also possible to relax the isolation property of two phase commit. If a RM receives a booking request while in the *Booking Held* state it can reject the request, advising the client that it is holding an uncommitted booking which may be released in the future. This prevents deadlock [19], when two users request the same resources at the same time.

The role of the TM can be played by the user but should be carried out by a trusted intermediary service.

VI. CO-ALLOCATION AND CONSENSUS

Co-allocation is a consensus problem. Consensus is concerned with how to get a set of processes to agree a value [26]. Distributed transaction commit, of which two phase commit is one approach, is a special case of consensus were all the processes must agree on either commit or abort for a transaction. In the case of co-allocation the RMs must agree to either accept or reject a purposed schedule.

Consensus is a well understood problem with over twenty five years of research [13, 26–28]. For example it has been shown that distributed consensus is impossible in an asynchronous system with a perfect network and just one faulty processor [13]. In an asynchronous system it is impossible to differentiate between a processor that is very slow and one that has failed. To handle faults requires either fault detectors or partial synchrony. Consensus is an important problem because it can be used to build replicas: start a set of deterministic state machines in the same state and use consensus to make them agree on the order of messages to process [25, 31].

Paxos is a well known fault tolerant consensus algorithm. We present only a description of its properties and refer the reader to the literature [28, 29, 31] for a full description of the algorithm. In Paxos a leader process tries to guide a set of acceptor processes to agree a value. Paxos will reach consensus even if messages are lost, delayed or duplicated. It can tolerate multiple simultaneous leaders and any of the processes, leader or acceptor, can fail and recover multiple times. Consensus is reached if there is a single leader for a long enough time during which the leader can talk to a majority of the acceptor processes twice. It may not terminate if there are always too many leaders. There is also a Byzantine version of Paxos [8] to handle the case were acceptors may have Byzantine faults.

Paxos Commit [18] is the Paxos algorithm applied to the distributed transaction commit problem. Effectively the transaction manager of two phase commit is replaced by a group of acceptors — if a majority of acceptors are available for long enough the transaction will complete. Gray and Lamport [18] showed that Paxos Commit is efficient and has the same message delay as two phase commit for the fault free case. They also showed that two phase commit is Paxos Commit with only one acceptor.

Though it may be possible to apply Paxos directly to the co-allocation problem by having the RMs act as acceptors we chose to use Paxos Commit instead. The advantage Paxos Commit has over using Paxos directly for co-allocation is that the role played by the RMs is no more complex than in the two phase approach. Limiting the role of the RM makes it more acceptable to resource providers and reduces the possibilities for faulty behaviour from the RM.

VII. HARC

The HARC approach to co-allocation is similar to the two phase approach described in Section II except the TM is replaced with a set of Paxos acceptors. The user sends a booking request to an acceptor who first replicates the message to the other acceptors using Paxos and then it uses Paxos Commit to make the reservations with the RMs. HARC terminates once it has decided to commit or abort a set of reservations. Users and RMs may modify or cancel reservations after HARC has terminated, but this is outside the scope of HARC.

The aims of HARC are deliberately limited. It does not address the issues of how to choose the op-

timal schedule; how to manage a set of reservations once they have been made; how to negotiate quality of service with the resource provider or how to support compensation mechanisms when a resource fails to fulfil a reservation or when a user cancels a reservation. HARC is designed so that other services and protocols can be combined with it to solve these problems.

A. Choosing a Schedule

The RM advertises the schedule when a resource may be available through a URI. The user retrieves the schedule using a HTTP GET. The information retrieved is for guidance only, the RM is under no obligation by advertising it. It is the RM's prerogative as to how much information it makes available, it may choose not to advertise a schedule at all. The RM may supply caching information along with the schedule using the cache support facilities of HTTP. The caching information can indicate when the schedule was last modified or how long the schedule is good for. The RM may also support conditional GET [11] so that clients do not have to retrieve and parse the schedule if it hasn't changed since the last retrieval. From the schedules for all the resources the user chooses a suitable time when the resources he requires might be free.

A co-scheduling service for HARC could use HTTP conditional GET to maintain a cache of resource schedules from which to create co-schedules for the user. The co-scheduling service would have to store only soft state making it easy to replicate.

B. Submitting the Booking Request

The user constructs a booking request which contains a sub-request for each resource that he wants to reserve. The booking request also contains an identifier chosen by the user, the UID. The combination of the identity of the user and the UID should be globally unique. The user sends the booking request to an acceptor using a HTTP POST. This acceptor will act as the leader for the whole booking process unless it fails in which case another acceptor will take over.

The leader picks a transaction identifier, the TID, for the booking request from a set of TIDs that it has been initially assigned. Each acceptor has a different range of TIDs to choose from to prevent two acceptors trying to use the same TID. The acceptor effectively replicates the message to the other acceptors by having Paxos agree the TID for the message. This instance of Paxos agrees a TID for the combination of the user's identity and the user chosen UID. If the user resends the message to the acceptor or to any other acceptor he will receive the same TID.

The user should continue submitting the request to any of the acceptors until he receives a TID — the submission of the booking request is idempotent across all the acceptors.

If a user reuses a UID he will get the TID of the previous booking request associated with that UID. To avoid reusing a UID the user can record all the UIDs he has previously used, however a more practical approach is to randomly choose a UID from a large namespace. Two users can use the same UID as it is the combination of the user's identity and the UID that is relevant.

The TID is a RequestURI [11]. The user can use a HTTP GET with the TID to retrieve the outcome of the booking request from any of the acceptors. The TID is returned to user using the HTTP Location header in the response to the POST containing the booking request. The HTTP response code is 201 indicating that a new resource has been created. The acceptor should return a 303 HTTP response code if a TID has already been chosen for the request to allow the user to detect the case when a UID may have been reused.

C. Making the Reservations

After the TID has been chosen the leader uses Paxos Commit to make the bookings with the RMs.

The booking request is broken down into the sub-requests and the sub-requests are sent to the appropriate RM using HTTP POST. The TID is also sent to the RM as the Referer HTTP header in the POST message.

The RM responds with a URI that will represent the booking local to the RM and whether the booking is being held or has been rejected. It also broadcasts this message to all the other acceptors.

As in the Paxos Commit algorithm the acceptors decide whether the schedule chosen by the user is acceptable to all the RMs or has been rejected by any of the RMs. The leader informs each RM whether to commit or abort the booking using a HTTP PUT on the URI provided by the RM.

It is the RM's obligation to discover the outcome of the booking. If the RM does not receive the commit or abort message it should use a HTTP GET with the TID on any of the acceptors to discover the outcome of the booking. Once the acceptors have made a decision it can be cached by intermediaries.

A HTTP GET on the TID returns the outcome of the booking request and a set of links to the reservations local to each RM. Detailed information on a reservation at a particular RM can be retrieved by

using a HTTP GET on the link associated with that reservation.

The user can cancel the reservation through the URI provided by the RM and the RM can advertise that it has cancelled the reservation using the same URI. The user should poll the URI to monitor the status of the reservation, HTTP HEAD or conditional GET can be used to optimize the polling.

There is a question of what happens if a RM decides to unilaterally abort from the *Booking Held* state which is not allowed in two phase commit or Paxos Commit. Since HARC terminates when it decides whether a schedule should be committed or aborted we can say that the RM cancelled the reservation after HARC terminated. This is possible because there is only a partial ordering of events in a distributed system [25]. The user can only find out that the RM cancelled the reservation after HARC terminated. The onus is on the user to monitor the reservations once they have been made and to deal with any eventualities that may arise.

D. HARC Actions

HARC supports a set of actions for making and manipulating reservations: Make, Modify, Move and Cancel. Make is used to create a new reservation, Modify to change an existing reservation (*eg* to change the number of CPUs requested), Move to change the time of a reservation and Cancel to cancel a reservation. A booking request can contain a mixture of actions and HARC will decide whether to commit or abort all of the actions. The HARC actions provide the user with flexibility for dealing with the case of a RM cancelling a reservation, for example he could make a new reservation on another resource at a different time and move the existing reservations to the new time.

A third party service could monitor reservations on the behalf of the user and deal with any eventualities that arise using the HARC actions in accordance to some policy provided by the user.

E. Security

The complete booking request sent to the acceptor is digitally signed [5] by the user with a X.509 [1] certificate. The acceptors use the signature to discover the identity of the user. Individual sub-requests are also digitally signed by the user so that the RMs can verify that the sub-request came from an authorized user. The acceptors also digitally sign the sub-requests before passing them to the RMs so that the RM can verify that the sub-request came from a trusted acceptor. If required a sub-request can be digitally encrypted [21] by the user so that the acceptors cannot read it.

A HTTP GET using the TID returns only the outcome of a booking request and a set of links to the individual bookings so there is no requirement for acceptors to provide access control to this information. The individual RMs control access to any detailed information on reservations they hold.

VIII. HARC AVAILABILITY

HARC has the same fault tolerance properties as Paxos Commit which means it will progress if a majority of acceptors are available. If a majority of acceptors are not available HARC will block until a majority is restored, since blocking is unacceptable we define this as HARC failing. To calculate the MTTF for HARC we use the notation and definitions from [19].

The probability that an acceptor fails is $1/MTTF$ were MTTF is the Mean Time To Failure of the acceptor. The probability that an acceptor is in a failed state is approximately $MTTR/MTTF$, were MTTR is the Mean Time To Repair of the acceptor.

Given $2F+1$ acceptors, the probability that HARC blocks is the probability that F acceptors are in the failed state and another acceptors fails.

The probability that F out of the $2F+1$ acceptors are in the failed state is:

$$\frac{(2F+1)!}{F!(F+1)!}\left(\frac{MTTR}{MTTF}\right)^F \tag{1}$$

The probability that one of the remaining $F+1$ acceptors fails is:

$$(F+1)\left(\frac{1}{MTTF}\right) \tag{2}$$

The probability that HARC will block is the product of (1) and (2):

$$\frac{(2F+1)!}{(F!)^2}\left(\frac{MTTR^F}{MTTF^{F+1}}\right) \tag{3}$$

The MTTF for HARC is the reciprocal of (3):

$$MTTF_{HARC} \approx \frac{(F!)^2}{(2F+1)!}\left(\frac{MTTF^{F+1}}{MTTR^F}\right) \tag{4}$$

Using 5 acceptors, $F=2$, each with a MTTF of 120 days and a MTTR of 1 day, $MTTF_{HARC}$ is approximately 57,600 days, or 160 years.

IX. BYZANTINE FAULTS AND HARC

HARC is based on Paxos Commit which assumes the acceptors do not have Byzantine faults but which

can happen in a Grid environment. We believe that acceptors can be implemented as fail–stop [19] services that are provided as part of a *Virtual Organization's* role of creating trust between organizations. Provision of trusted services is one way of realizing the *Virtual Organization* concept. If Byzantine fault tolerance is necessary then Byzantine Paxos [8] could be used in HARC.

If a HARC acceptor does have a Byzantine fault it cannot make a reservation without a signed request from a user. HARC has been designed to deal with Byzantine fault behaviour from users and RMs.

No co-allocation protocol can guarantee that the reserved resources will actually be available at the scheduled time. HARC is a fault tolerant protocol for making a set of reservations, it does not attempt to make any guarantees once the reservations have been made. The user may be able to deal with the situation were a resource is unavailable at the scheduled time by booking extra resources that can act as backup. This is an application specific solution and again illustrates the importance of end-to-end arguments [35]. HARC provides a fault tolerant service to the user but fault tolerance is still necessary at the application level.

X. HARC/1

HARC/1, an implementation of HARC, has been successfully demonstrated at SuperComputing 2005 and iGrid 2005 [20] where it was used to co-schedule ten compute and two network resources. HARC/1 is implemented using Java with sample RMs implemented in Perl. HARC/1 is available at `http://www.cct.lsu.edu/personal/maclaren/CoSched`.

XI. CONCLUSION

As the size of a distributed system increases so to does the probablity that some component in the system is faulty. However, if a fault tolerant approach is applied then increasing the size of the system can mean that the reliability of the overall system improves. Just as Beowulf systems built out of commodity components have displaced expensive supercomputers so clusters of PCs are displacing expensive mainframe type systems [15]. HARC demonstrates how important services can be made fault tolerant to create a fault tolerant Grid.

HARC has been demonstrated to be a secure, fault tolerant approach to building co-allocation services. It has also been shown that the user and RM roles in HARC are simple. The state–transitions for the RM in HARC are the same as for the two phase approach illustrated in Fig. 1. The only extra requirement for the RM over the two phase approach is that it must broadcast a copy of its *Booking Held* or *Abort* message to all acceptors.

The use of HTTP contributes to the simplicity of HARC. HTTP is a well understood application protocol with strong library support in many programming languages. HARC demonstrates through its use of HTTP and URIs the effectiveness of REST as an approach to developing Grid services.

HARC's functionality can be extended by adding other services, for example to support co-scheduling and reservation monitoring. Extending functionality by adding services, rather than modifying existing services, indicates good design and a scalable system in accordance to REST.

The opaqueness of the sub-requests to the acceptors means that HARC has the potential to be used for something other than co-allocation. For example a HARC implementation could be used as an implementation of Paxos Commit to support distributed transactions.

XII. ACKNOWLEDGEMENTS

The authors would like to thank Jim Gray, Savas Parastatidis and Dean Kuo for discussion and encouragement. The work is supported in part through NSF Award #0509465, "EnLIGHTened Computing".

[1] C. Adams and S. Farrell. Internet X.509 Public Key Infrastructure Certificate Management Protocols. IETF RFC 2510, 1999.

[2] R. Allan *et al.* Building the e-Science Grid in the UK: Grid Information Services. Proceedings of UK e-Science All Hands Meeting. 2003.

[3] A. Andrieux *et al.* Web Services Agreement. Draft GGF Recommendation, September 2005.

[4] D. Baker and M. Mc Keown. Building the e-Science Grid in the UK: Providing a software toolkit to enable operational monitoring and Grid integration. Proceedings of UK e-Science All Hands Meeting. 2003.

[5] M. Bartel *et al.* XML-Signature Syntax and Processing. W3C Recommendation, February 2002.

[6] T. Berners-Lee, R. Fielding and L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. IETF RFC 3986, 2005.

[7] R. Blake *et al.* The teragyroid experiment—supercomputing 2003. Scientific Computing,

13(1):1-17, 2005.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance. Proceedings of 3rd OSDI, New Orleans, 1999.

[9] K. Czajkowski et al. A protocol for negotiating service level agreements and coordinating resource management on distributed systems. Proceedings of 8th International Workshop on Job Scheduling Strategies for Parallel Processing, eds D Feitelson, L. Rudolph and U. Schwiegelshohn, Lecture Notes in Computer Science, 2537, Springer Verlag, 2002.

[10] K. Czajkowski *et al.* Grid Information Services for Distributed Resource Sharing. Proceedings of the Tenth IEEE International Symposium on High-Performance Distributed Computing (HPDC-10), IEEE Press, 2001.

[11] R. Fielding *et al.* Hypertext Transfer Protocol – HTTP/1.1, IETF RFC 2616, 1999.

[12] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. PhD thesis. University of California, Irvine, 2000.

[13] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. Journal of the ACM, 32(2), 1985.

[14] I. Foster, C. Kesselman and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International J. Supercomputer Applications, 15(3), 2001.

[15] S. Ghemawat, H. Gobioff, and S. Leung. The Google Filesystem. 19th ACM Symposium on Operating Systems Principles, Lake George, NY, October, 2003.

[16] M. Gudgin *et al.* SOAP Version 1.2 Part 1: Messaging Framework. W3C Recommendation, June 2003.

[17] J. Gray. Notes on data base operating systems. In R. Bayer, R. Graham, and G. Seegmuller, editors, Operating Systems: An Advanced Course, volume 60 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Heidelberg, New York, 1978.

[18] J. Gray and L. Lamport. Consensus on Transaction Commit. Microsoft Research Technical Report MSR-TR-2003-96, 2005.

[19] J. Gray and A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[20] A. Hutanu *et al.* Distributed and collaborative visualization of large data sets using high-speed networks. Submitted to proceedings of iGrid 2005, Future Generation Computer Systems. The International Journal of Grid Computing: Theory, Methods and Applications, 2005.

[21] T. Imamura, B. Dillaway and E. Simon. XML Encryption Syntax and Processing. W3C Recommendation, December 2002.

[22] I. Jacobs and N. Walsh. Architecture of the World Wide Web, Volume One. W3C Recommendation, December 2004.

[23] S. Jha *et al.* Spice: Simulated pore interactive computing environment – using grid computing to understand dna translocation across protein nanopores embedded in lipid membranes. Proceedings of the UK e-Science All Hands Meetings, 2005.

[24] D. Kuo and M. Mc Keown. Advance reservation and co-allocation for Grid Computing. In First International Conference on e-Science and Grid Computing, volume e-science, IEEE Computer Society Press, 2005.

[25] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. Communications of the ACM 21, 7, 1978.

[26] L. Lamport, M. Pease and R. Shostak. Reaching Agreement in the Presence of Faults. Journal of the Association for Computing Machinery 27, 2, 1980.

[27] L. Lamport, M. Pease and R. Shostak. The Byzantine Generals Problem. ACM Transactions on Programming Languages and Systems 4, 3, 382-401, 1982.

[28] L. Lamport. The Part-Time Parliament. ACM Transactions on Computer Systems 16, 2, 133-169, 1998.

[29] L. Lamport. Paxos Made Simple. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001) 18-25, 2001.

[30] B. Lampson. Hints for computer system design. ACM Operating Systems Rev. 17, 5, pp 33-48, 1983.

[31] B. Lampson. How to build a highly available system using consensus. In Distributed Algorithms, ed. Babaoglu and Marzullo, Lecture Notes in Computer Science 1151, Springer, 1996

[32] E. Rescorla. HTTP Over TLS. IETF RFC 2818, 2000.

[33] L. Roberts. Resource Sharing Computer Networks. IEEE International Conference, New York City. 1968.

[34] A. Roy. End-to-End Quality of Service for High-end Applications. PhD thesis. University Of Chicago, Illinois, 2001.

[35] J. Saltzer, D. Reed and D. Clark. End-to-end arguments in system design. Proceedings of the 2nd International Conference Distributed Computing Systems, Paris, 1981.

[36] D. Skeen. Nonblocking commit protocols. In SIGMOD '81: Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data. ACM Press, 1981.

[37] TeraGrid. `http://www.teragrid.org/`.

[38] UK National Grid Service. `http://www.ngs.ac.uk/`.

[39] K. Yoshimoto, P. Kovatch and P. Andrews. Co-scheduling with user–settable reservations. In Job Scheduling Strategies for Parallel Processing, eds E. Frachtenberg, L. Rudolph and U. Schwiegelshohn, Lecture Notes in Computing Science, 3834, Springer, 2005.