Effects of an Asynchronous Resource Allocation Protocol on End-to-End Service Provision

Spyros Lalis, Manolis Marazakis, Dimitris Papadakis
Computer Science Department University of Crete and Institute of Computer Science FORTH
{lalis,maraz,dimpapa}@ics.forth.gr

Abstract

In this paper we present results of experiments that were performed to quantify the effects of retries in an asynchronous resource allocation protocol that is employed for end-to-end service provision. A major finding from our experiments is that an application class may be penalized by experiencing delays in accessing a shared resource as a result of overload on resources used by other application classes, which are unknown to this class. In turn, these delays may cause under-utilization of other resources used by this application class. Moreover, we illustrate how such effects emerge simply by varying the relative occurrence frequencies of application classes, but without changing the overall request arrival process. It is shown that such changes can lead to undesirable performance effects for all classes that share a number of resources.

Keywords: *multi-resource allocation, electronic trading, autonomous open systems, workload dynamics, end-to-end service provision*

1. Introduction

With the number of resources and users connected to the Internet rising rapidly, we are about to witness a shift from best-effort systems to architectures that support QoS. Typical applications, which already require end-to-end QoS to achieve a satisfactory performance, are video-on-demand and teleconferencing. In the near future, many more applications with similar resource allocation requirements will emerge, such as telemedicine, advanced digital libraries, globally distributed computing, real-time remote laboratories and monitoring stations, and virtual classrooms. Thus, it will be often the case that a single application involves multiple services, requiring coordinated allocation of several different resources.

It is unlikely that a single authority will ever be responsible for managing the vast number of resources made available through the global network. Resources will be controlled by independent service-providers, as this is the case today for bandwidth and web server capacity. However, the autonomy of resource providers limits the control that external entities have over the resources owned and managed by these authorities. This, in turn, imposes restrictions on coordinated resource allocation, thereby making it a potentially complex process.

To relieve applications from this task, special *middleware* services called resource brokers can act as intermediaries for the resource offerings of multiple resource manager authorities, and take upon themselves the responsibility to provide their clients (applications) with the resources that they require in a bundled fashion. In other words, brokers access each resource manager independently, and, following an acquisition strategy, reserve resource access with the client-specified QoS guarantees at each service point.

In this paper we investigate the performance-related effects of multi-resource allocation in a dynamic and open environment, using our prototype of a market-based brokerage service [LaNiPaMa98]. Assuming complete autonomy of the resource providers, we do not allow direct access to their internal allocation tables nor direct locking of their resources. Instead, resource acquisition is done on a simple trial-and-

error basis, until all the resources required for application execution are acquired successfully.

A major finding from our experiments is that an application class may be penalized by experiencing delays in accessing a shared resource as a result of overload on resources used by other application classes, which are unknown to this class. In turn, these delays may cause under-utilization of the other resources used by this application class. Moreover, we illustrate how such effects emerge simply by varying the relative occurrence frequencies of application classes, but without changing the overall request arrival process. It is shown that such changes can lead to undesirable performance effects for all classes that share a number of resources.

The rest of the paper is organized as follows. Section 2 briefly describes the system configuration with focus on the acquisition protocol employed to coordinate allocation of resource bundles. The experiments that have been conducted using this system are presented in Section 3. In Section 4, our work is put in perspective with respect to a broader classification of resource acquisition protocols, pointing out future research directions. Section 5 gives an overview of related work, identifying parallels and differences with our approach. Finally, Section 6 concludes the paper.

2. System Architecture

The experiments are conducted using a prototype system. The system has several independent primitive services with guaranteed quality of service, which exhibit reliable performance behavior. It also features a brokerage mechanism for negotiating allocation of resources at the various underlying services according to application requirements.

The configuration used for the experiments comprises three main components: the session manager, the broker, and the QoS managers. Figure 1 shows the system architecture. A brief description of these components is given in the following.

2.1 The QoS Managers

The QoS managers are independent entities controlling the resources of services. Each service is represented via its own QoS manager. In order for a service to be accessible, the corresponding QoS manager must first register with the broker. This is done via a *sell* request, which contains information about the service type and the QoS supported. A service can remove itself from the system by letting its QoS manager send a *cancel* request to the broker. There is no limitation regarding the arrival and departure of services, thus the system is truly *open*.

To invoke a service, a request along with a description of the desired performance characteristics is sent to its QoS manager. Before accepting a request, the QoS manager verifies that the desired service can be provided at the requested quality. If this is not possible, the corresponding request is rejected.

In the configuration used for the experiments presented in this paper, there are three QoS managers, one for the network (there is a single network provider) and one for each server (two different servers are used). Servers are stream transmitters that continuously send data, at a rate specified by the application. The network is an ATM switch connecting the server machines to client workstations.

2.2 The Session Manager

The session manager is primarily responsible for converting application requests into primitive service requests. Also, the quality/performance terms of applications are mapped into QoS specifications for each of the requested services. Having identified the required service types, the session manager constructs a list containing the services (resources) required to implement the application request, and sends it to the broker in order to determine the actual service instances to be used. Upon receiving a response, the session manager invokes the selected services in the appropriate order.

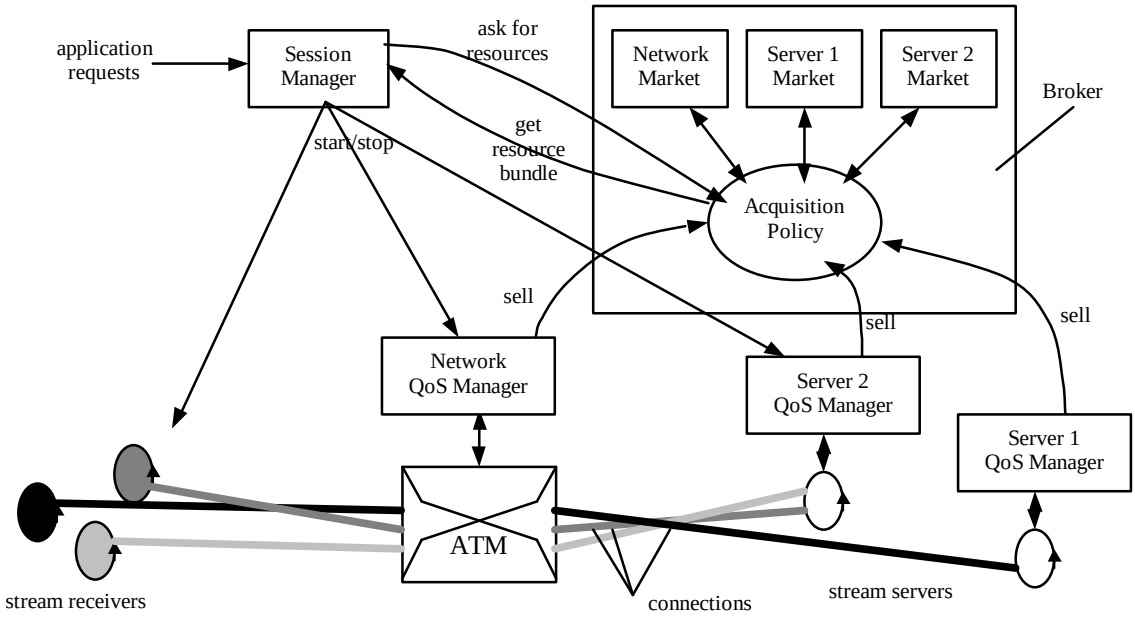


Figure 1. The system architecture

In our experiments, the applications running on top of the system are elementary. A single request type is introduced, demanding that data is generated by a stream server and transferred over the network, at a certain rate and for a given duration. Hence, the session manager converts each application request into a pair of orders, one for a server and one for the network, and forwards this list to the broker. In order for the application request to be serviced, both orders have to be satisfied simultaneously. Then, a connection is created and data transmission is initiated with calls to the corresponding components. When the specified duration elapses, transmission is stopped and the connection is closed.

Each application is endowed with a budget for acquiring resources, which reflects its priority relative to other applications. Since we focus on the resource acquisition policy, rather than the competition among different priority classes, in our experiments, all requests have the same budget.

2.3 The Broker

The broker acts as an intermediary between the session manager seeking service instances of a certain type and the QoS managers advertising the processing capabilities of underlying services. It encapsulates the resolution process that is activated to provide the session manager with the list of concrete service instances to be used for a particular application request. Binding application tasks to resources is dynamic, allowing the architecture to accommodate variations in service availability.

The broker uses an *economic paradigm* to match offerings and demand for a given service type. Service providers are *sellers* advertising their resources to applications. Applications are *buyers* requesting resources that they purchase from the sellers. Sell and buy orders are matched using *continuous double auction*, which allows buyers to make offers, and sellers to accept those offers, at *any* particular moment. The market price is determined by the interplay of supply and demand and is always between the highest buy and the lowest sell price. Trading takes place within autonomous market objects that are dynamically created as orders and offers regarding various resources arrive at the broker. In our case, there are three markets, one for the network and one for each server.

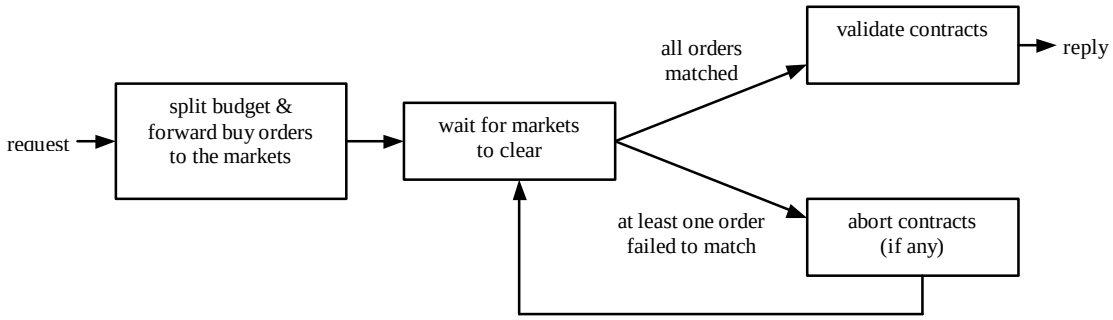


Figure 2. The Resource Acquisition Protocol

Since the markets operate *independently* from each other, there is a chance for applications requesting both resources to get a match in one market and fail to do so in the other. In order to achieve the desired bundling of resources without imposing any synchronization among the underlying resource providers, the broker employs the following non-blocking, iterative acquisition protocol (Figure 2):

1. For each incoming request, the budget is equally divided among the resource types needed. By this simple strategy each application has a predetermined highest value to bid for each resource. Each buy order is then forwarded to the corresponding market in order to be matched against available service offers.
2. When the markets clear, they return to the broker a list of contracts, i.e. pairs of matched sell and buy orders. These are inspected to determine which requests can be satisfied atomically, i.e. whether respective matches have been established for the server and the network.
3. If both buy orders of a request have been matched successfully, the corresponding contracts are validated by sending an acknowledgement to the markets. Also, a reply containing the addresses of the service providers to be invoked is sent to the session manager (as a response to its request).
4. Else, if only one buy order has been matched, the respective contract is aborted and the buy order is *retried*, i.e. it is re-issued in the market. Notably, the other buy order remains *pending*. If both buy orders of a request are pending, nothing needs to be done. Then, the broker waits until the next clearing of the markets, i.e. the bundling process restarts from step 2.

Pending orders are results of overload. On the contrary, *retries* occur as a side effect of the asynchronous acquisition protocol. Retries not only impose a processing

overhead to the system; they may also result in lost opportunities for the service providers and thus lower utilization of their resources.

3. Experimental Observations

We investigate the performance-related effects of uncoordinated multi-resource allocation in a dynamic and open environment, using our prototype of a market-based resource allocation service described in the previous section. Our aim is to quantify the effects of canceling contracts for resources on end-to-end service provision, for workloads consisting of application classes with overlapping resource requirements.

A simple system configuration and workload scenario where two application classes share a network resource while each accesses a different class-specific computation/data server is used. It turns out to capture essential performance-related aspects on dynamic and open environments, and serves to highlight the effects caused by resource dependencies among application classes. These dependencies are studied by correlating measurements collected by independent resource managers. This allows us to accurately track the sequence of events leading to interference from one class' resource consumption pattern to the resource consumption pattern of another.

Furthermore, we study the impact of changes in the workload due to variations in the relative occurrence frequency of application classes. Such changes, which are common in open systems, affect the load level of class-specific resources and, through the dependencies among classes due to shared resources, may lead to undesirable performance for all classes.

3.1 Configuration and Performance Parameters

The system configuration is defined, in accordance to the model of our prototype as presented in the previous section, by the capacity of the computation/data servers and of the network connecting client applications with the servers. Two application classes with overlapping resource requirements are introduced. Both classes require access to the shared network resource NET. Class C1 requires access to server SRV1, while class C2 requires accesses to server SRV2.

For both application classes, each client requests one “capacity-slot” on each of the resources. The capacity of NET is assumed to be 30 slots, while the capacity of each of SRV1 and SRV2 is assumed to be 15 slots. As described in Section 2.3, for each of the resources NET, SRV1 and SRV2 there is an independent market. Since we ignore the issue of resource pricing, all resources are priced in the same way, and applications are given the same budget.

Each request, regardless of the application class that it belongs to, occupies resource slots for a duration that is uniformly distributed in the interval from 8 to 10 seconds. Thus the average service duration for a request is 10 seconds. Further, the inter-arrival time between requests, which determines the request arrival process, is exponentially distributed with a mean value of 0.4 seconds, leading to an average of 2.5 arrivals per seconds. It should be noted that the exponential distribution results in occasional "load spikes", during which the number of arrivals per second can be much higher than the average. This behavior models bursts in the request arrival processes of open systems. Finally, the time interval between successive clearings of the resource markets is set to 0.1 seconds.

We report results for two different settings for the relative occurrence frequencies of classes C1 and C2, so as to enable an evaluation of the impact of changes in the

quality of the workload. In the first setting, the relative frequencies of C1 and C2 are 60% and 40%, respectively, and in the second setting, the relative frequencies of C1 and C2 are 70% and 30%, respectively. While in both cases the aggregate arrival rate remains unchanged, there is a crucial difference in the relative loads of the class-specific resources SRV1 and SRV2. In the 60-40 workload both SRV1 and SRV2 can sustain the incoming load, whereas in the 70-30 workload SRV1 becomes overloaded. Notably, NET can sustain the aggregate load in both workload mixes.

3.2 Performance Metrics

In the experiments presented in the following, the load generator of our prototype forwarded to the session manager a series of 500 requests, according to the aforementioned workload characteristics.

The behavior of the system is studied by collecting measurements of the number of pending requests and the number of retries, for each application class and resource. These measurements are collected by the broker at the end of each market clearing, after determining the resource acquisition contracts that have to be cancelled. The results are presented as time series for successive clearings of the resource markets. Thus, in all subsequent figures, the horizontal axis represents successive points in time when market clearing took place, and the vertical axis represents the number of retried (bold line) and pending (gray line) requests.

Furthermore, we measure the time required to acquire all resources (setup delay), for each application class. The setup delay is a function of the average pending requests, but it is also affected by the average number of retries for each resource, per class. Therefore we explicitly report statistics on the number of retries experienced for each class and resource.

3.3 Alternative Policies for Resource Acquisition Retries

As described in Section 2.3, the broker cancels resource acquisition contracts when it cannot simultaneously acquire slots for all the resources required for servicing a client request. Then, it attempts to acquire the resources needed from scratch, by re-issuing the cancelled buy order. Hence, there is a question of how to resolve among retried and new buy orders.

An obvious choice, designated from now on as "Policy A", is for retried buy orders be handled without any differentiation from buy orders originating from new request arrivals. Under this policy, a resource market selects randomly among the *equivalent* buy orders accumulated at each clearing period. In this context, "equivalent" means equal number of requested resource slots and offered price per slot. This policy is expected to result in quite high setup delays for requests that involve retries, as under high load these requests may suffer additional delays due to competition with an ever-increasing number of requests. This effect is quantified in Section 3.4.

An alternative to "Policy A" is to differentiate retried from new buy orders. Under this policy, designated from now on as "Policy B", resource markets select among buy orders in chronological order, giving retried orders precedence over new buy orders. This policy favors application classes that experience overload on their class-specific resources, as their requests will get increased priority the longer they remain pending and the more they get retried. The effects of this policy on the number of pending and retried requests per market clearing are examined in Section 3.5.

Policy A is studied in order to provide a comparison baseline for the more elaborate policy B. It also reveals the dependencies among application classes due to resource sharing and demonstrates the adverse performance effects of uncoordinated multi-resource allocation. In Section 3.6, we compare the two policies presented in this section in terms of setup delay, maximum and mean number of retries per request, for each class and resource.

3.4 Measurements for Policy A

Figures 3 and 4 depict the time series for the numbers of pending and retried requests for C1 and C2, respectively, for the 60-40 workload under Policy A. It can be seen that there are hardly any retries for C2, whereas (after a short load build-up period) C1 exhibits retries.

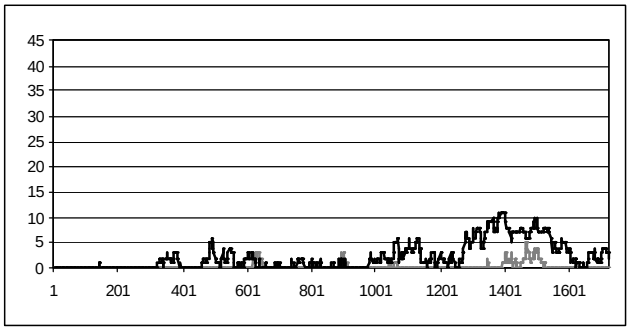


Figure 3. C1-Requests (Policy A / 60-40 load)

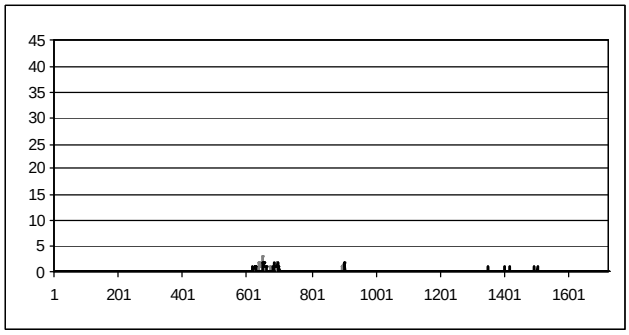


Figure 4. C2-Requests (Policy A / 60-40 load)

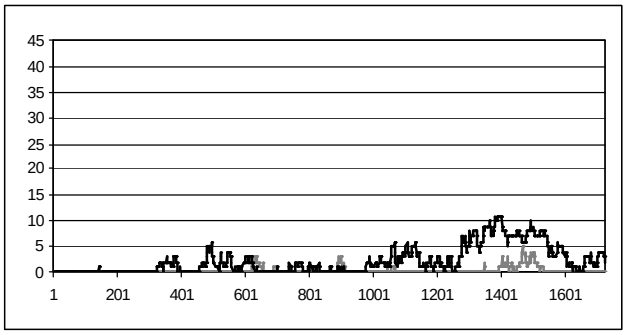


Figure 5. NET C1-Requests (Policy A / 60-40 load)

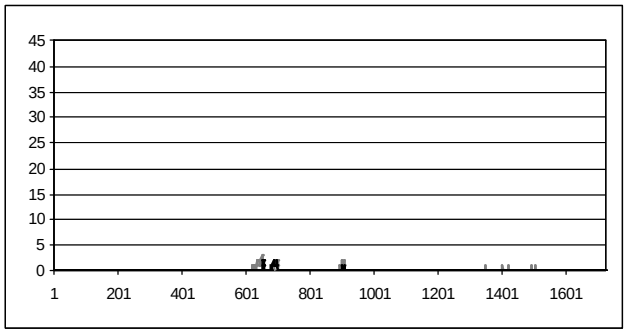


Figure 6. NET C1-Requests (Policy A / 60-40 load)

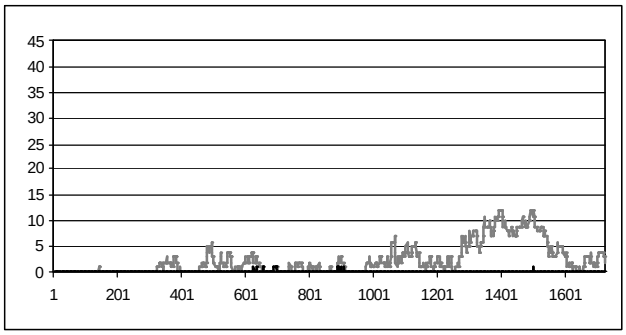


Figure 7. SRV1 C1-Requests (Policy A / 60-40 load)

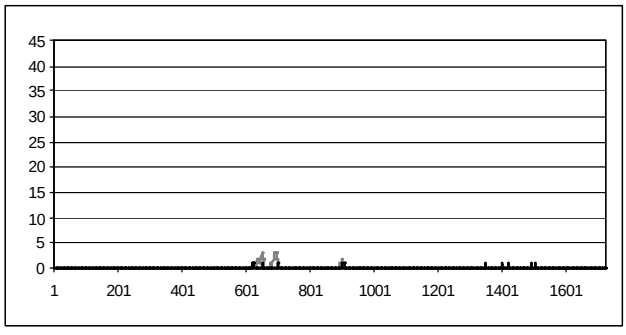
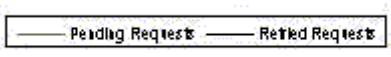


Figure 8. SRV2 C1-Requests (Policy A / 60-40 load)



As shown in Figure 5, these retries are mainly due to the retries for NET resource. The reason for these retries is that, although C1 requests can easily acquire NET slots, they occasionally fail to acquire SRV1 slots, because its capacity is marginally sufficient to sustain the incoming load. Thus, the broker has to cancel NET resource

acquisition contracts for C1 requests. The number of pending requests for SRV1, shown in Figure 7, indirectly drives the NET retry curve.

The effect is much less noticeable for C2, since there is ample capacity for the class-specific resource SRV2. Nevertheless, Figure 6 shows that there are occasional pending C2 requests for NET slots, which cause corresponding retries for the SRV2 resource (see Figure 8). This is because frequent C1 retries for NET cause an increase on the number of requests competing for NET slots. This is reflected by pending requests for NET of C1 shown in Figure 5, as well as C2 requests shown in Figure 6.

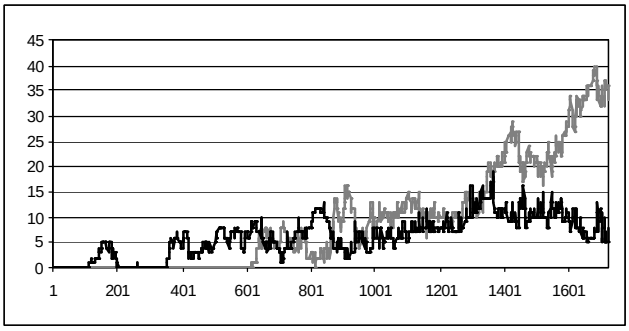


Figure 9. C1-Requests (Policy A / 70-30 load)

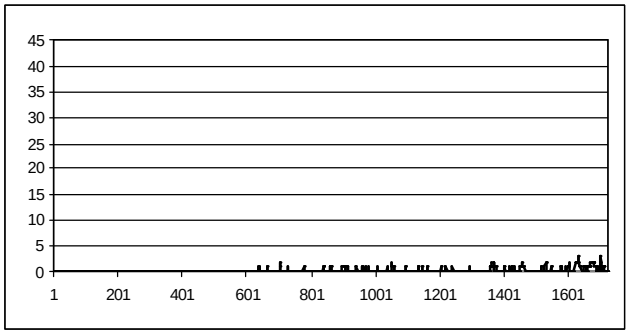


Figure 10. C2-Requests (Policy A / 70-30 load)

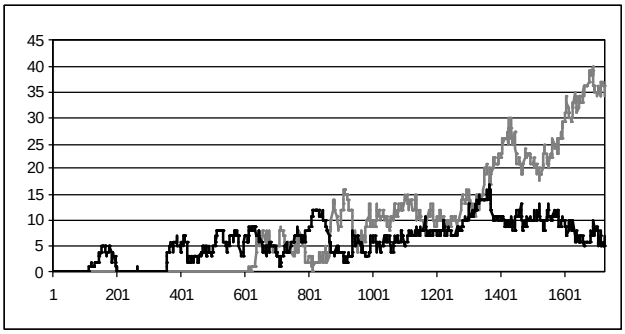


Figure 11. NET C1-Requests (Policy A / 70-30 load)

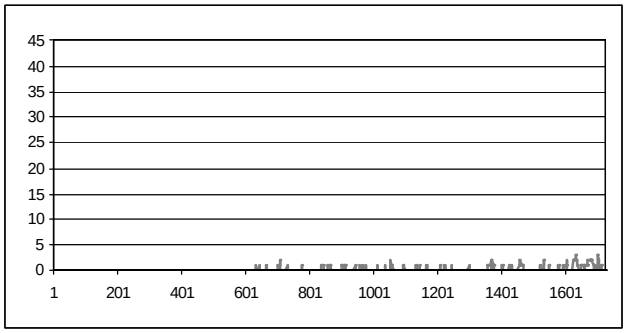


Figure 12. NET C2-Requests (Policy A / 70-30 load)

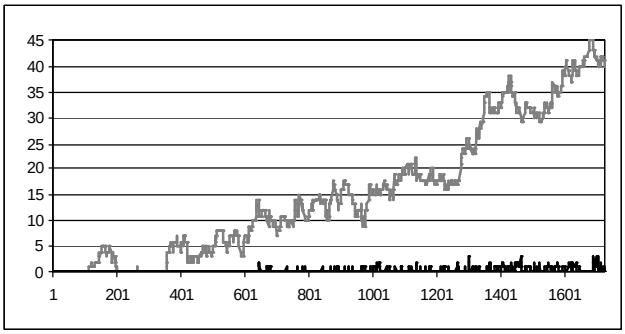


Figure 13. SRV1 C1-Requests (Policy A / 70-30 load)

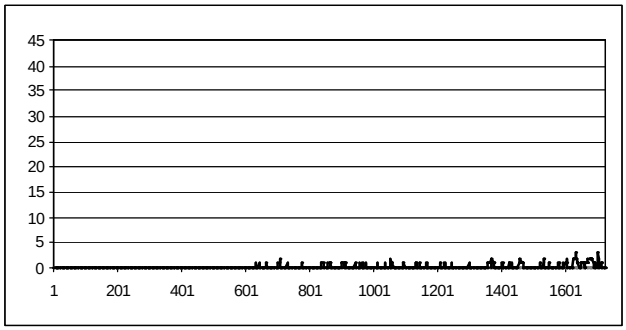
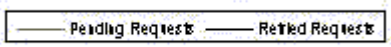


Figure 14. SRV2 C2-Requests (Policy A / 70-30 load)



For the 70-30 workload, these phenomena become more pronounced, as shown in Figures 9 and 10 for classes C1 and C2, respectively. The number of pending C1 requests rises without bound, because there is insufficient capacity for the SRV1 resource to sustain the incoming load (see Figure 13). Therefore, the effect that pending C1 requests for SRV1 cause retries for NET is much more visible (Figure 11). The large number of C1 retries for NET increase competition, leading to a

noticeable number of C2 pending requests for NET slots, as shown in Figure 12. In turn, this results in retries for SRV2 (Figure 14).

As with the 60-40 workload, C2 requests, although occasionally penalized as a result of increased competition on the shared resource (NET), are not significantly affected by the high load imposed on the SRV1 resource. This is a consequence of the fact that policy A does not discriminate against new arrivals in favor of retries. Thus, C2 requests are given equal opportunity to access the shared resource NET as the C1 requests that often get retried.

3.5 Measurements for Policy B

Figures 15 and 16 show the time series for the numbers of pending and retried requests for classes C1 and C2, and the 60-40 workload under policy B. For this workload, there is no major difference between policies A and B. However, as can be seen by comparing Figures 3 and 15, policy B results in a slight decrease on the number of pending and retried requests for the C1 class.

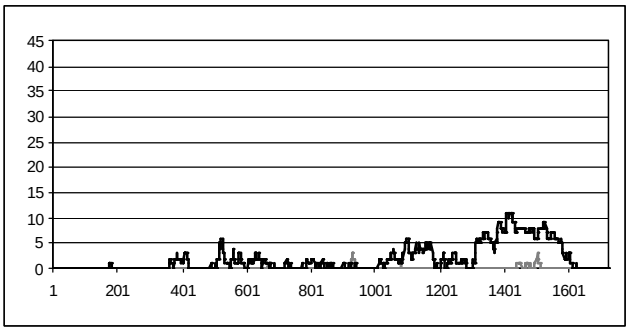


Figure 15. C1-Requests (Policy B / 60-40 load)

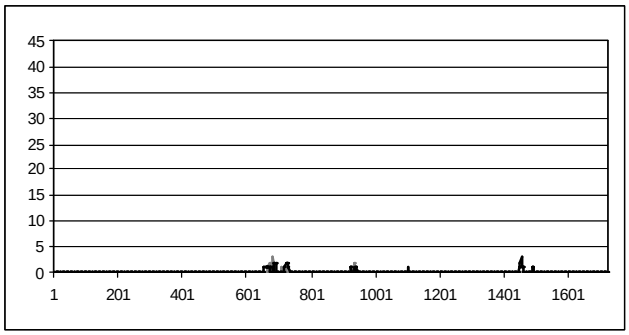


Figure 16. C2-Requests (Policy B / 60-40 load)

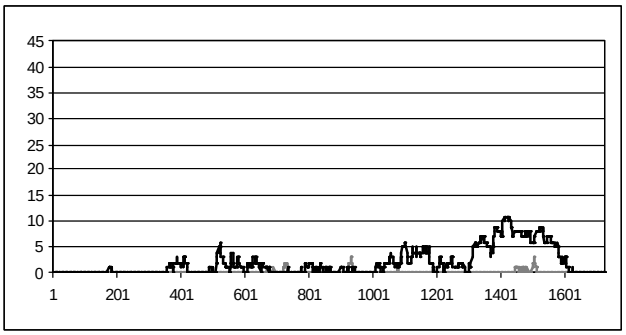


Figure 17. NET C1-Requests (Policy B / 60-40 load)

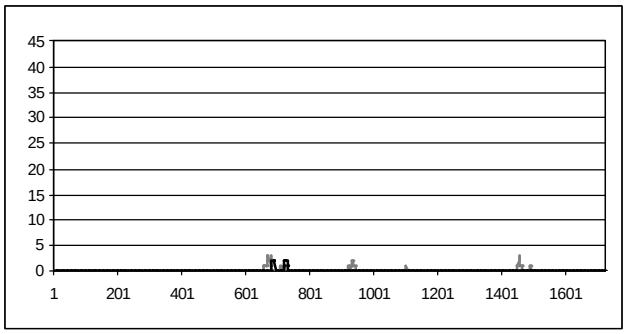


Figure 18. NET C2-Requests (Policy B / 60-40 load)

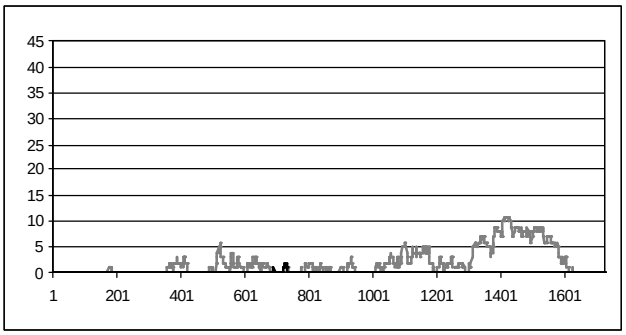


Figure 19. SRV1 C1-Requests (Policy B / 60-40 load)

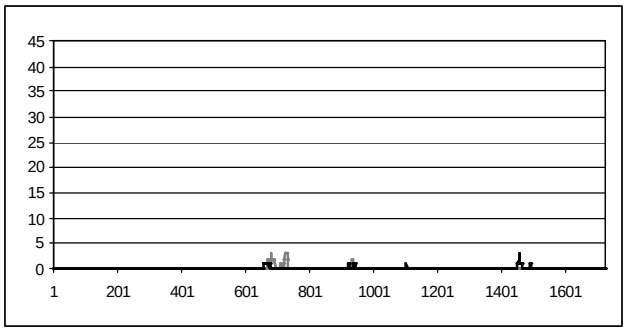
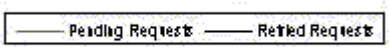


Figure 20. SRV2 C2-Requests (Policy B / 60-40 load)



This can be attributed to a corresponding decrease on the number of pending and retried C1 on the NET resource (see Figures 17 and 18). Notably, this occurs at the expense of C2, which (as seen by comparing Figures 6 and 18) exhibits a slight increase on the number of pending requests on the NET resource. This increase on the number of pending requests on the NET resource causes a corresponding increase on the number of retries for the SRV2 resources, as the broker cancels SRV2 contracts for class C2.

These effects become much more evident under the 70-30 workload, as shown in Figures 21 and 22. From Figure 25 it can be seen that there are no retries for SRV1 slots by class C1 requests, unlike the situation under policy A (Figure 13). Moreover,

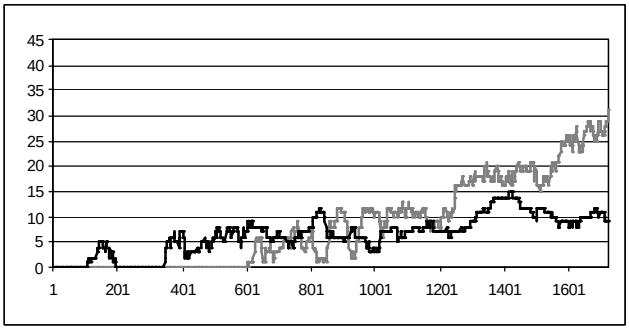


Figure 21. C1-Requests (Policy B / 70-30 load)

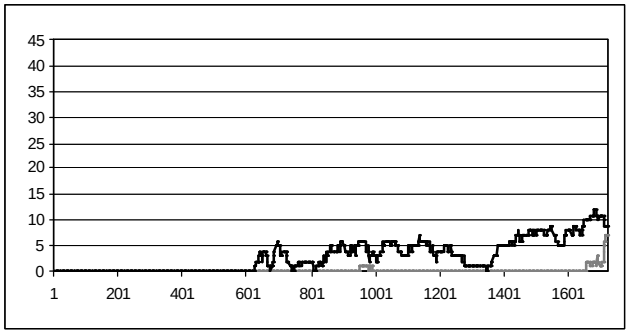


Figure 22. C2-Requests (Policy B / 70-30 load)

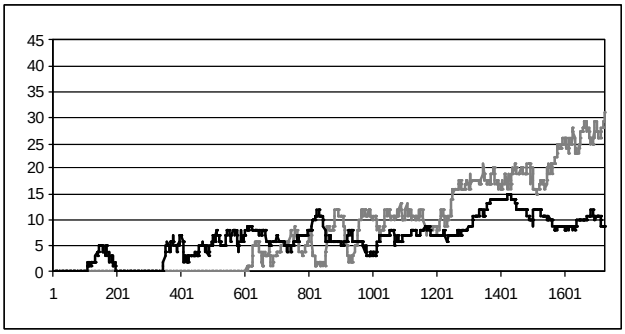


Figure 23. NET C1-Requests (Policy B / 70-30 load)

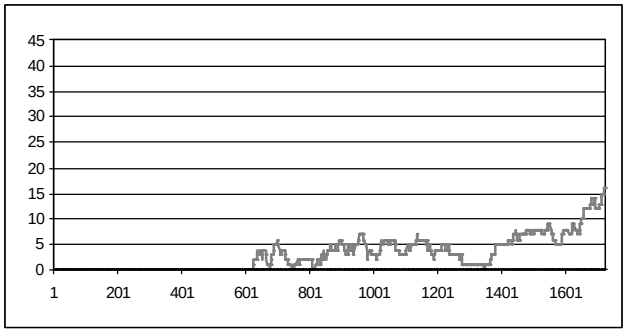


Figure 24. NET C2-Requests (Policy B / 70-30 load)

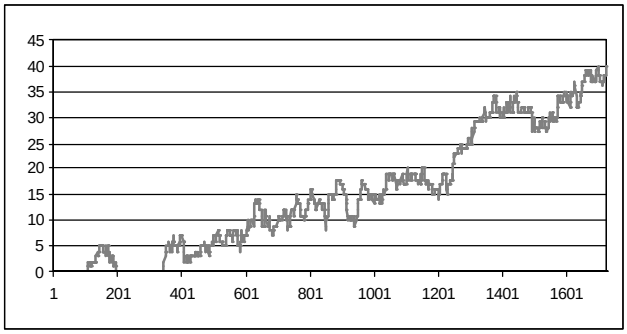


Figure 25. SRV1 C1-Requests (Policy B / 70-30 load)

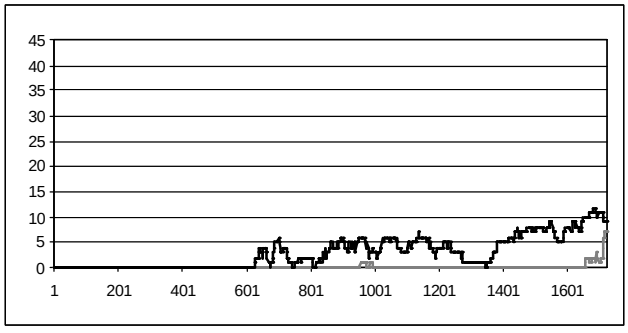
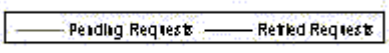


Figure 26. SRV2 C2-Requests (Policy B / 70-30 load)



the number of pending requests for NET slots, depicted in Figure 23, is lower than under policy A (Figure 11). Nevertheless, SRV1 remains overloaded and the number of pending requests for its slots rises without bound, as seen in Figure 25.

However, this slight improvement for class C1 is achieved at the expense of class C2. This is mainly because it becomes more difficult for C2 requests to acquire NET slots,

as it can be inferred from Figure 24. This difficulty is due to the fact that (after a brief load build-up phase) C1 requests remain much longer in the pending state in the NET market, which under policy B increases the likelihood that they will be selected. Since C1 requests find it increasingly difficult to acquire SRV1 slots as well, a fraction of NET slots allocated to C1 requests are wasted due to retries. In other words, the retries for NET slots by C1 requests are given increased priority under policy B, resulting in an increased handicap for the C2 class.

As the C1 backlog grows, this problem escalates and results in a growing number of retries for C2 requests. The overload on SRV1 results in an ever-growing fraction of the C1 requests alternating between being unable to obtain NET slots and acquiring only NET slots but then having to release them as they cannot acquire corresponding SRV1 slots. This cycle increases the amount of time that C1 requests spend in the system and the expected number of retries. As a byproduct of increased delay in acquiring NET slots for C2 requests, new arrivals create a backlog for SRV2 slots as well, as shown in Figure 26.

Thus policy B, with its preference to requests that have spent considerable time in the system waiting for resource slots and that have experienced retries, breaks down in the case of overload for resource SRV1. In fact, it penalizes class C2 that has no direct dependence on this resource, and, remarkably, despite the fact that the shared resource NET has sufficient capacity to sustain the aggregate load. Since there is no other distinction between the classes other than their differing (but overlapping) resource requirements, from the system's point of view it would be more beneficial for NET slots that are wasted by the C1 class to be allocated to C2 requests.

It is important to note that this anomaly remains hidden under the 60-40 workload, where policy B appears to have a similar performance impact as policy A. However, a slight change in the relative occurrence frequency of the two application classes suffices to unveil this problem and to destabilize the system. Most important of all, none of the authorities managing the resources for NET and SRV2 can explain (or address effectively) this problem in isolation.

3.6 Comparisons of Setup Delay, Maximum and Mean Number of Retries

In this section we present a further comparison of policies A and B in terms of the mean setup delay and statistics for the number of retries, for each application class and resource. Table 1 shows the mean setup delay for classes C1 and C2 under policies A and B, for both workloads. It also shows the mean setup delay as measured over all requests regardless of class. Tables 2 and 3 show the mean and the maximum number of retries per class and resource, for both workloads.

	Class C1		Class C2		Overall	
Policy/Workload	**60-40**	**70-30**	**60-40**	**70-30**	**60-40**	**70-30**
A	1.96	9.01	0.16	0.38	1.26	9.01
B	1.55	11.48	0.20	6.65	1.03	10.07

Table 1. Mean Setup Delay (in seconds)

In the 60-40 workload, policy B results in lower setup delays as measured over all requests (decrease of 18%) in comparison with policy A. This is achieved by a decrease of setup delay for class C1 (by 20%), and a corresponding increase of setup delay for class C2 (by 25%). The overall decrease is attributed to the fact that more C1 requests are issued than C2 requests. In the 70-30 workload, policy B actually increases the mean setup delay over all requests by approximately 12%, while the

mean setup delay for classes C1 and C2 is increased by 27% and 1650%, respectively. While for the 60-40 workload, an increase of the setup delay for class C2 is justified by a corresponding improvement of the setup delay for class C1, in this case class C2 suffers a tremendous performance penalty without any improvement for class C1. This is due to the instability problem highlighted in Section 3.5.

	Class C1 NET		Class C1 SRV1		Class C1 – Overall	
Policy/Workload	**60-40**	**70-30**	**60-40**	**70-30**	**60-40**	**70-30**
A	14.28 / 146	35.37 / 251	0.08 / 7	1.00 / 9	14.37 / 146	36.37 / 259
B	12.29 / 191	35.75 / 76	0.05 / 4	0 / 0	12.34 / 195	35.75 / 76

Table 2. Mean/Maximum Number of Retries for the C1 class

	Class C2 – NET		Class C2 - SRV1		Class C2 – Overall	
Policy/Workload	**60-40**	**70-30**	**60-40**	**70-30**	**60-40**	**70-30**
A	0.17 / 9	0 / 0	0.17 / 5	1.63 / 20	0.35 / 9	1.63 / 20
B	0.20 / 5	0 / 0	0.53 / 21	42.40 / 114	0.74 / 26	42.40 / 114

Table 3. Mean/Maximum Number of Retries for the C2 class

For the 60-40 workload, policy B decreases the mean number of retries for class C1 requests (by 14%) while increasing it for class C2 requests (by 111%). For the NET resource, the decrease in the mean number of retries for class C1 is 14%. This is matched by a 17% increase for class C2 on NET. The breakdown of policy B, in the 70-30 workload, for class C2 is evident. The mean number of retries for C1 requests decreases by less than 2%, leading to a corresponding increase of 2500% for C2 requests. There are no retries for the NET resource by C2 requests because C2 requests are pending, waiting for NET slots to be released (see Figure 24). Likewise, there are no retries for SRV1 by C1 requests, but as shown in Figure 25 the number of pending requests is growing without bound.

It is interesting to see that the maximum number of retries under policy B is consistently higher than under policy A for the 60-40 workload, and consistently lower for the 70-30 workload.

4. Discussion and Future Work

From our experiments it becomes clear that retries are not merely an internal processing overhead of the brokerage mechanism, but may indeed lead to lost opportunities for service providers, thus resulting in lower utilization of their resources, and performance degradation for some applications.

Notably, retries can be avoided without enforcing a strict synchronization scheme among the underlying resource providers. Instead of letting the broker issue a buy order to all markets in parallel and wait for matches, it is possible to issue a buy order in one market only, and wait for this order to be matched before issuing a buy order to the other market. In analogy to the nomenclature of group-oriented communication, the former protocols could be termed as "multicast" protocols while the latter could be called a "ring" protocol.

The ring protocol is particularly attractive when the order in which buys are sent to the markets is chosen so that unnecessary contention for shared resources is explicitly avoided. As an example, for the configuration used in Section 3, it is better to acquire server capacity first, before attempting to acquire network capacity, since SRV1 is the bottleneck resource.

Indeed, a series of preliminary experiments shows that the ring protocol completely eliminates the retries both for the C1 and C2 class. For this reason, the choice of the retry policy is also irrelevant for the ring protocol. However, this method comes with its own weaknesses, as it can be inferred from Table 4 that gives the mean setup delay.

	Class C1		Class C2		Overall	
Policy/Workload	**60-40**	**70-30**	**60-40**	**70-30**	**60-40**	**70-30**
Multicast - A	1.96	9.01	0.16	0.38	1.26	9.01
Multicast - B	1.55	11.48	0.20	6.65	1.03	10.07
Ring A/B	2.40	12.50	0.24	0.18	1.56	8.91

Table 4. Mean Setup Delay of the Ring Protocol

The setup delay of the ring protocol is smaller only for class C2 and the 70-30 load, where the retry probability is high when using a multicast protocol. However, the performance of class C1 slightly deteriorates, because apart from the delay caused by the overload on SRV1 the sequential nature of the acquisition method imposes an additional delay. The overall setup delay nevertheless decreases, because the gains for the C2 class carry more weight. For the 60-40 load both classes exhibit longer setup delays with the sequential method because of the extra acquisition overhead, even though there is no overload.

It is important to note that both protocols can be viewed as special cases of a more general, tree-structured protocol family, with nodes indicating the protocol (multicast or ring) employed for acquiring a particular group of "abstract" resources, themselves also being nodes (Figure 27). The leaves of the tree are resource groups consisting of a single resource type.

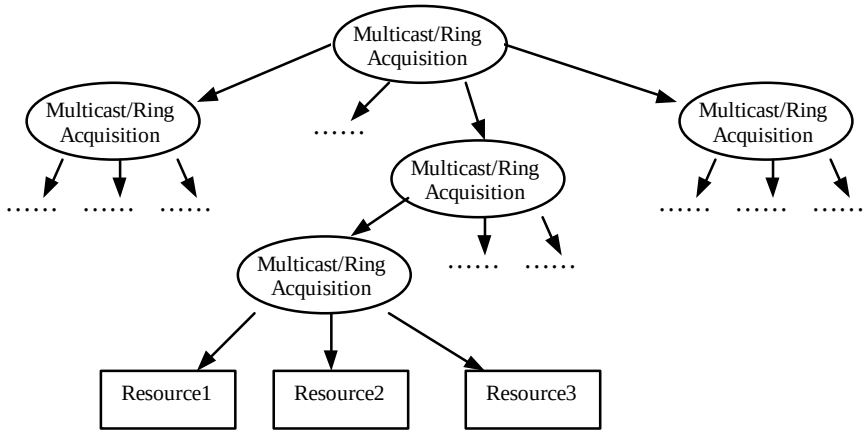


Figure 27. The general, resource acquisition protocol family

Using this classification, the acquisition protocol studied in this paper is a tree protocol of depth 1 using a multicast method to acquire two resources. Analogously, the sequential method introduced in this section, is a tree protocol of depth 1 using a ring method to acquire these resources.

Identifying the key parameters of the multicast and ring methods is important since this will enable a rudimentary, offline performance analysis of the entire protocol family. It must be noted, however, that in an open and dynamic system it cannot be determined a priori which instantiation of the tree protocol should be used to acquire the resources needed by a certain application class. Furthermore, in the case of the

ring protocol, the order at which it is attempted to acquire a given set of resources also cannot be specified in advance. The reason is that these decisions depend not only on the resources required by each application class but also on the overall application load relative to the capacity of the underlying services, at the time when a request enters the system. It is thus interesting to compare the problem of choosing the appropriate acquisition protocol to distributed query optimization problems that have been extensively addressed in the past, and for which working and well-tested solutions already exist [OzVa91].

5. Related Work

Our system employs market-based control to allocate resource bundles to applications using auctions. *Market-oriented programming* has been proposed as a general solution technique for resource allocation [We95]. Also, auctions of various types have been applied to a wide range of resource allocation problems arising in distributed systems [Cl95]. The WALRAS environment [ChWe97] has been developed for the simulation of computational economies. The Michigan AuctionBot [WuWeWa98] uses configurable components to support parameterized auctions.

The prototype presented is a specialization of a broader resource allocation framework developed in the context of our ongoing work towards dynamic service composition in an open environment [MaPaNi97]. There are related designs that have parallels with our approach.

The Auction Manager [MuWe98], developed in the context of the University of Michigan Digital Library project, is a middleware service designed to support creation of auctions, matching of agents to auctions, and notification of agents upon creation of new auctions that match their interests. Such a service could be used by our brokerage mechanism to locate market objects. Since we are considering allocation of low-level resource "slots" from two functionally different resources, however, the problem of locating appropriate markets is considerably simplified, compared to higher-level information services in a digital library.

Most of the literature related to resource allocation for distributed systems have been concerned with allocation of a single “item”, such as processing time [WaHoHuKeSt92], communication bandwidth [Fe89], access to data [Fe93] or network information services [MuWe95]. Our work is concerned with allocating multiple resources in a single "bundle", as this is the type of resource allocation problem arising in the realization of composite services in open environments.

In [HaBoDs95], a negotiation architecture is introduced, which allows to return in response to user requests a set of present and future proposals for resource contracts, instead of a simple acceptance or rejection signal. An important difference is that this kind of negotiation can be done only if the expected service duration of applications is known in advance. Also, there is a direct coupling of the entity responsible for the coordinated resource allocation and the individual resources. In our system, the broker has no knowledge of the request duration, and resource selection is done indirectly via the market mechanism.

Further, [CoJaMoGi97] presents an architecture to support a variety of commerce transaction types, from simple direct buying and selling to complex multi-agent contract negotiations. In this case, markets for specific commodities are hosted in "exchanges", which are network-accessible resources that support a set of markets and provide common services. Market services are delivered to a participating agent

through a "market session", which encapsulates the state of all interactions in the process of contracting with other agents. This corresponds to the broker component of our architecture, which handles the details of establishing resource bundles on behalf of applications. However, no concrete resource acquisition strategy is studied.

In another design, presented in [RaRiDiCh97], individual resources use a calendar metaphor for arranging their schedule. A client locks a time slot when it has committed to using the associated resource during the period denoted by the slot. Clients implement a two-phase commit protocol [Gr79] to atomically reserve slots for all the resources that they require. In our implementation, service providers are assumed to be *autonomous*, and the broker cannot synchronize the individual markets to allocate resource bundles. The ring protocol discussed in Section 4 can introduce a similar reservation effect, but the broker has no control over the actual resolution process among orders and offers for each resource. It is thus impossible to perform global optimization.

Our experimental results indicate that the asynchronous and uncoordinated operation of multiple independent auctions poses non-trivial problems in achieving efficient allocation of resource bundles. In this respect, our work complements related work such as [WaWe98], which considers convergence properties of bidding strategies and auctions in a setting involving multiple items. They further consider running the auctions simultaneously, using the WALRAS simulation environment for multi-agent systems. Our prototype demonstrates some of the implementation issues arising in a real distributed environment.

6. Conclusions

A major result of this experimental study is that overload on one resource may cause another resource to remain underutilized. An application class may be penalized, by experiencing increased delays in accessing a shared resource as a result of overload on other resources, unknown to it.

It is important to consider this finding in the light of effects due to changes in the overall workload. Such changes are a common occurrence in open systems, cannot be predicted a-priori due to the dynamic nature of the environment. In particular, we study a case where the relative frequencies of requests belonging to different application classes change in such a way that the aggregate request arrival process remains unchanged. This type of changes affects the load level of class-specific resources, which may in turn lead to undesirable performance effects for all classes due to cancelled partial contracts for resources shared among classes.

This study points out that issues of stability have to be explicitly addressed in multi-resource allocation, as the dynamics of open environments may lead to undesirable performance for all application classes if there is no strict coordination among the resource managers. It is important to stress that in open environments it may be impossible - or even undesirable - to synchronize autonomous resource managers.

References

[ChWe97] J.Q. Cheng and M.P. Wellman. "The WALRAS Algorithm: A Convergent Distributed Implementation of General Equilibrium Outcomes". *Computational Economics,* to appear. Available via URL http://auction.eecs.umich.edu, 1997.

[Cl95] S.H. Clearwater, editor. "Market-Based Control: A Paradigm for Distributed Resource Allocation". World Scientific, 1995.

[CoJaMoGi97] J. Collins, S. Jamison, B. Mobasher, M. Gini. "A Market Architecture for Multi-Agent Contracting". *Technical Report* 97-15, University of Minessota, 1997.

[Er96] J. Eriksson, N. Finne, and S. Janson. Information and Interaction in MarketSpace – Towards an Open Agent-based Market Infrastructure". *In Proceedings of the USENIX Workshop on Electronic Commerce,* 1996.

[Fe89] D.F. Ferguson, C. Nikolaou, and Y. Yemini. "An Economy for Flow Control in Computer Networks". In *Proceedings of IEEE Infocom '89,* pages 110-118, Washington, DC, 1989. IEEE Computer Society Press.

[Fe93] D.F. Ferguson, C. Nikolaou, and Y. Yemini. "An Economy for Managing Replicated data in Autonomous Decentralized Systems". In *Proceedings of ISADS'93, International Symposium in Autonomous Decentralized Systems,* pages 367-375, Los Alamitos, CA, 1993. IEEE Computer Society Press. Kawasaki, Japan.

[Gr79] J. Gray. "Notes on Database Operating Systems". In *Operating Systems: An Advanced Course*. Springer-Verlag, 1978.

[HaBoDs95] A. Hafid, G.v. Bochmann and R. Dssouli. "Quality of Service Negotiation with Present and Future Reservations: A Detailed Study". *Technical Report 983, University of Montreal,* 1995.

[Hu88] B.A. Huberman. "The Ecology of Computation". North-Holland, 1988.

[LaNiPaMa98] S. Lalis, C. Nikolaou, D. Papadakis, M. Marazakis. "Market-Driven Resource Allocation in a QoS-capable Environment". In Proceedings of ICE'98, *International Conference on Computational Economies,* ACM Press, 1998.

[MaPaNi97] M. Marazakis, D. Papadakis, and C. Nikolaou. "The Aurora Architecture for Developing Network-Centric Applications by Dynamic Composition of Services". *Technical Report TR 213, FORTH/ICS,* 1997.

[MuWe95] T. Mullen and M.P. Wellman. "A Simple Computational Market for Network Information Services". In *Proceedings of the Int'l Conference on Multiagent Systems,* June 1995. San Fransisco,

CA.

[MuWe98] T. Mullen and M.P. Wellman. “The Auction Manager: Market Middleware for Large-Scale Electronic Commerce”, In *Proceedings of the Workshop on Agent-Mediated Electronic Trading (in conjunction with the 2nd Int’l Conf. on Autonomous Agents)*, 1998.

[OzVa91] M. Ozsu and P. Valduriez. “Principles of Distributed Database Systems”. Prentice Hall, 1991.

[RaRiDiCh97] R. Ramamoorthi, A. Rifkin, B. Dimitrov, and K.M. Chandy. ``A General Resource Reservation Framework for Scientific Computing''. In *Proceedings of the 1st Int'l Scientific Computing in Object-Oriented Parallel Environments (ISCOPE) Conference*, 1997.

[St97] D.O. Stahl. “The Inefficiency of Auctions in Dynamic Stochastic Environments”. *Working Paper 9709, Center for Applied Research in economics, University of Texas-Austin*, 1997.

[WaHoHuKeSt92] C.A. Waldspruger, T. Hogg, B.A. Huberman, J. Kephart, and S. Stornetta. “Spawn: A Distributed Computational Ecology”. *IEEE Transactions on Software Engineering*, 18(2), 1992.

[WaWe98] W.E. Walsh and M.P. Wellman. “A Market Protocol for Distributed Task Allocation”. In *Proceedings of the Int'l Conference on Multiagent Systems*, 1998.

[We95] M.P. Wellman. “Market-oriented programming: Some early lessons”. In S.H. Clearwater, editor, *Market-Based Control: A paradigm for distributed resource allocation*, chapter 4, pages 74-95. World Scientific, 1995.

[WuWeWa98] P.R. Wurman, M.P. Wellman, and W.E. Walsh. ``The Michigan Internet AuctionBot: A Configurable Auction Server for Human and Software Agents''. In *Proceedings of the Int'l Conference on Autonomous Agents*, 1998.