DIMACS Technical Report 2003-15
April 2003

Towards an Efficient Cluster-based E-Commerce Server

by

Victoria Ungureanu[1]
Department of MSIS
Rutgers Business School - Newark and New Brunswick
Rutgers University
180 University Ave., Newark, NJ 07102
email:ungurean@rbs.rutgers.edu

Michael Katehakis Department of MSIS
Rutgers Business School - Newark and New Brunswick
Rutgers University
180 University Ave., Newark, NJ 07102
email: mnk@andromeda.rutgers.edu

Benjamin Melamed
Department of MSIS
Rutgers Business School - Newark and New Brunswick
Rutgers University
94 Rockafeller Rd., Piscataway, NJ 08854
email: melamed@rbs.rutgers.edu

[1]Work supported in part by DIMACS under contract STC-91-19999, and Information Technology and Electronic Commerce Clinic, Rutgers University

---

ABSTRACT

Cluster-based server architectures combine good performance and low cost, and are commonly used for applications that generate heavy loads. Essentially, a cluster-based server consists of a front-end dispatcher and several back-end servers. The dispatcher receives incoming requests, and then assigns them to back-end servers, which are responsible for request processing. The many benefits of cluster-based servers make them a good choice for e-commerce applications as well. However, applying this type of architecture to e-commerce applications is hindered by the fact that e-commerce clusters have the additional task of verifying that requests comply with contract terms. The problem is further complicated by the fact that contract terms may be expressed as functions of dynamic, mutable states.

The problem addressed in this paper is the effective assignment of e-commerce requests, such that the load is balanced among back-end servers and request validation is efficient. To this end, we propose a policy called `TDA` (Type Dependent Assignment), which takes account of the type of contracts. Under `TDA`, stateless contracts are replicated on all back-end servers. In contrast, a stateful contract, `C`, is preassigned to a designated back-end server, called the base of `C`, which is responsible for maintaining the state of `C`. The operation of `TDA` may be broadly outlined as follows. A request governed by a stateless contract is assigned to the least loaded server. In contrast, a request governed by a stateful contract is assigned to its base, if the base is not overloaded, and to the least loaded server, otherwise.

To gauge the efficacy of `TDA`, the paper presents an empirical study, using real-life trace data measured at Internet cluster-based servers. Experimental results of this study show that `TDA` outperforms the `Least Connected` policy by a factor of four. To support `TDA` implementation, we propose a contract formulation, which explicitly states the contract type, and which distinguishes between the mutable and immutable parts of a contract. We experimentally show that the proposed contract formulation lends itself to efficient enforcement.
Keywords: stateful contract, assignment policy, enforcement

1 Introduction

Motivated by the need to cut costs and increase competitiveness, more and more enterprises are migrating to on-line transactions with trading partners [5]. Among the problems inherent in this migration, none is more serious than the difficulty of controlling the activities of the disparate agents involved in e-commerce.

Trading relations between enterprises are based on mutually agreed contracts. Generally, contracts enumerate agents authorized to participate in transactions, and spell out such things as the rights and obligations of each partner and the terms and conditions of trades. Moreover, at any given time an enterprise may be bound by several contracts simultaneously. For example, Ford has approximately thirty thousand suppliers, each operating under a different contract, and General Motors has about forty thousand [6] (both companies have recently announced their intention to perform their inter-enterprise business on-line).

As an example of a contract, consider that agents in a client enterprise (say Ford) may purchase merchandise (tires, in this example) from a supplier enterprise (say Firestone), under the following terms:

- *Firestone honors purchase orders (POs) issued by Ford, for up to a cumulative value, called* blanket*, of 10,000 tires each month at a prescribed price of $25/tire.*
- *Only agents duly certified as purchase officers by* `ford_CA` *(a designated certification authority of the client enterprise) may issue POs; only agents duly certified as sale representatives by* `firestone_CA` *(a designated certifying authority of the supplier enterprise) are authorized to respond to POs.*

Contract terms may take into account dynamic information, referred to as *state*. The state, which *evolves* over the lifetime of the contract, might consist of various contract phases, the state of transactions regulated by the contract, or past actions of various agents. A contract whose terms are expressed in terms of state is called *stateful*; a contract that consists only of immutable terms is called *stateless*. For example, the contract presented above is stateful, and its state is the value of the blanket. The validity of a PO depends on the current value of the blanket: a PO is valid provided the number of tires requested does not exceed the current value of the blanket. Furthermore, the blanket value is decreased whenever a valid PO is issued.

The satisfactory execution of e-commerce applications depends on the performance of the corresponding servers. To date, applications that generate heavy loads, commonly use *cluster-based* server architectures [2, 7, 11] that combine good performance and low cost. Essentially, a cluster-based server consists of a front-end dispatcher and several back-end servers, as depicted in Figure 1. The dispatcher receives incoming requests, and then assigns them to back-end servers, which are responsible for request processing. This type of architecture achieves transparency and a certain degree of scalability and fault-tolerance by decoupling request receiving from processing.

The many benefits of cluster-based servers make them a good choice for e-commerce applications. However, applying this type of architecture to e-commerce is hindered by the fact that an e-commerce cluster needs to verify that requests comply with the governing contracts. To emphasize the inherent difficulties of this problem, we consider several straightforward solutions and discuss their pitfalls.

First, consider replicating all contracts on all back-end servers. This solution has the potential advantage of load balancing, because a request can be assigned to the least loaded server with the assurance that that server can verify compliance. However, it may lead to poor performance, because contract states have to be maintained consistent. Unfortunately, achieving consistency through traditional algorithms, such as two-phase commit, would strain back-end servers: *every change* of the state of *every contract* would have to be committed by *all back-end servers*, even though they might never serve a request regulated by some of the contracts.

Secondly, assume that requests governed by a given contract are assigned only to a pre-specified back-end server. Because a contract is served by only one back-end server, this solution does not require maintaining the consistency of contract states. However, it might lead to load imbalance.

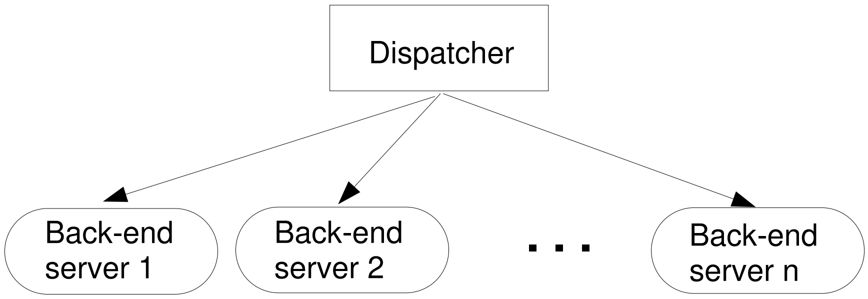


Figure 1: A cluster-based server

Finally, we mention that delegating request validation to the dispatcher (or some other central entity, for that matter) defeats the purpose of using a distributed architecture in the first place.

In order to implement cluster-based architectures in e-commerce setting, we propose a policy that combine good load balancing with efficient validation of requests. The proposed policy, called `TDA` (Type Dependent Assignment), takes account of contract types. Under `TDA` stateless contracts are replicated on all back-end servers. In contrast, any stateful contract `C`, is preassigned to a designated back-end server $\mathtt{B}_{\mathtt{C}}$, called the *base* of `C`, which is responsible for maintaining the state of `C`.

The broad outline of `TDA` can be described as follows. For every incoming request, the dispatcher determines the type of contract governing it (stateless or stateful). If the contract is stateless, the dispatcher assigns the request to the least loaded server (determined according to a prescribed load metric). However, if the contract is stateful, the dispatcher assigns the request to the base of `C`, $\mathtt{B}_{\mathtt{C}}$, whenever the base is not overloaded. Note, that since the base maintains the state of `C`, no extra communications are needed in this case. In contrast, if the base is heavily loaded, the dispatcher reassigns the base to the least loaded server. We point out that only in this case, the state is sent to the new base and communication overhead is incurred. To support `TDA` implementation, we propose a contract formulation, which explicitly states contract type, and which distinguishes between mutable and immutable parts of a contract.

To gauge the efficacy of the proposed policy, the paper presents an empirical study using real-life trace data. Experimental results show that `TDA` outperforms traditional policies by a factor of four. Moreover, we show experimentally that the proposed contract formulation lends itself to efficient enforcement.

The rest of the paper is organized as follows. Section 2 presents the contract formulation which is illustrated by an example. Section 3 describes the contract enforcement mechanism and studies its performance. Section 4 describes in detail the `TDA` policy, and Section 5 presents a performance analysis for `TDA`, based on a simulation driven by empirical data. Section 6 discusses related work, and Section 7 concludes the paper.

2 Contract Formulation

Contracts represent the prevalent way for expressing expected rules of conduct of a group of people involved in a certain (economic) activity. In the following we will define our concept of contract, which we call *communication contract* (`CC`) in order to distinguish it from the general term[1]. Our goals are to mirror as closely as possible the social notion of contracts, while tailoring it for the specific needs of electronic commerce.

Specifically, a communication contract has two parts: first, a declarative part specifying the contract *type*

[1]Due to space limitations, our presentation of communication contracts and the following example is perhaps more terse than it should be.

and the initial value of the state — if the contract is stateful. Second, an executable part which consist of an explicit set of *rules* embodying the contractual terms. We will further elaborate on the components of a `CC`; to illustrate its structure and the nature of its rules we use the blanket contract introduced previously.

Type Each contract contains a clause of the form:

```
type(T)
```

where `T` denotes the state of the contract and which can have the values `stateful` or `stateless`. Additionally, a stateful contract defines the initial value of the state, using a statement of the form:

```
initialState(S)
```

For example, the blanket contract contains the clause `type(stateful)`. Its state consists of a term `blanket(b)`, where `b` represents the current number of tires left in the blanket. And the initial state of this contract is defined by the clause `initialState(blanket(10000))`.

Rules Contract terms can be quite naturally expressed by means of any formal language supporting *event-condition-action* (ECA) kind of rules. We are using here an extension of a language devised for support of control policies [8, 9] built on top of Prolog [3]. But the nature of this language is, in a sense, of a secondary importance.

In this language contract terms are embedded in rules of the form:

```
eval(R,Cert):-
      condition-1,...,condition-k,
       provision-1,...,provision-n,
       do(accept).
```

This rule states that, if *condition-1* through *condition-k* are satisfied then the request is valid. These conditions may take into consideration the request and the certificates presented. Moreover, the rule calls for carrying out *provision-1* through *provision-k*, denoting any additional actions required by the contract.

In addition to the standard types of Prolog goals, the body of a rule may contain a distinguished type of goal, called a *do-goal*. A do-goal has the form `do(p)`, where `p` is a *primitive-operation*. A sample of primitive operations is presented in Figure 2.

3 Contract Enforcement

The enforcement mechanism makes the following assumptions. First, it assumes that each back-end server has an additional component, called observer. Observers, are generic tools that can verify certificates, and interpret and carry out any well formed set of rules. Second, it assumes that the immutable part (type and rules) of all contracts, regardless of their type, is replicated on all back-end servers of the cluster.

Given these assumptions we are in position now to explain how the enforcement is carried out. Consider that the dispatcher receives a request `R`, and assigns it to a back-end server `S`, having an associated observer `O`. Then the following steps will be taken:

Operations on certificates	
`issuer(pk,cert)`	binds `pk` to the public key of the issuer of certificate `cert`;
`role(r,cert)`	binds `r` to the value of attribute `role` in `cert`, if one exists; fails otherwise;
Operations on the state	
`t@`	returns true if term `t` is present in the state, and fails otherwise
`incr(t(v),d)`	increments the value of the parameter `v` of term `t` with quantity `d`
`dcr(t(v),d)`	decrements the value of the parameter `v` of term `t` with quantity `d`
Miscellaneous	
`accept`	denotes that the request is valid;
`reject`	denotes that the request is invalid;

Figure 2: Some primitive operations.

- Observer `O` determines the contract `C` governing request `R`. For example, if it is a put request, the URL may serve as reference to the contract id.
- If the request is accompanied by certificates then, each certificate is verified, and translated into an internal format—essentially a Prolog list.
- The observer checks whether there is a rule in `C` authorizing the request. If there is, the processing cycle of `R` continues; otherwise, the request is discarded.

To insure that the state of a contract `C` is maintained consistent, requests regulated by `C` are handled *sequentially* in chronological order of their occurrence. The observer evaluates the contract and carries out this ruling *atomically*, so that the sequence of operations that constitute the ruling for one request does not interleave with those of any other request governed by the same contract `C`. Note that an observer might serve as base for several contracts, in which case requests governed by different contracts may be evaluated concurrently.

Experimental Results We have implemented in Java a toolkit which has been used to measure the performance of the observer. As a benchmark for observer performance we measured the average time taken by an observer to compute and carry out the ruling for a given request. Specifically, we have used purchase orders sent under the blanket contract. The measured processing time, averaged over 3000 runs, is 3.3 ms. In the experiment the server was running on a SUNW, Ultra-2 machine operating at 296 MHz, using Solaris 2.6 operating system and Java 1.2.

The current implementation of the toolkit is experimental and much less efficient than it can be. But even in its present state, it is quite affordable: the overhead due to rules evaluation is comparable with the time required for certificate verification—`20 ms`[2]—and smaller then communication time (typically `40ms` in a WAN).

[2]The average time for verification was computed in the same configuration as above; revocation information was given by expiration time, thus not requiring getting revocation information from a different server. Generally this time could grow boundlessly, depending on the availability of revocation infrastructure.

4 The TDA Policy

The TDA policy makes the following assumptions and stipulations. First, TDA stipulates that the initial choice of bases by the dispatcher is made in Round-Robin manner. Second, TDA assumes that at any given time, the dispatcher knows the number of requests pending completion at each back-end server. These values are estimated by the number of active server connections and are used as a measure of server loads. This is a practical assumption, because the dispatcher is responsible for handling connections and passing incoming data from clients to back-end servers. Consequently, the dispatcher must keep track of open and closed connections at each server (cf., for e.g., [11, 15]). Finally, TDA assumes that the dispatcher has a parameter, AC, defined as the maximum number of requests (expressed as the number of active connections) that a server may process concurrently (to avoid excessive delays).

Subject to the assumptions and stipulations above, TDA operates on a request R governed by a contract C as follows:

1. *When* R *arrives at the dispatcher, the dispatcher determines the type of* C*.*
2. *If* C *is stateless, the dispatcher assigns* R *to the least loaded the server, where the load of a server is given by its number of active connections.*
3. *If* C *is stateful, the dispatcher determines the base* S *of* C*. If the number of active connections of* S *is smaller than* AC*, then the request is assigned* S*.*
4. *However, if the number of active connections of* S *exceeds* AC*, the dispatcher determines the least loaded back-end server, say* S'*. If* S' *is less loaded than* S*, then the dispatcher marks* S' *as the base of* C *and assigns the request to it. The dispatcher further notifies* S *of the change. After* S *finishes processing all requests in its queue, governed by* C*, it sends the state of* C *to* S'*.*

We point out that base reassignment is motivated by the fact that the initial assignment is made in Round-Robin manner, which is only a crude means for load balancing. Even though all back-end servers act as bases for the same number of contracts, some of them may get an unusually high number of "active" contracts, which govern a large fraction of the requests received in a certain time interval, and consequently their performance may degrade.

Changing bases incurs communication overhead, because the new assignment needs to be communicated to the old and new bases, and the state needs to be transferred to the new base. To balance the benefit of base change with its cost, the dispatcher does not change the base every time there is a server which is less loaded than the base. Rather, a base reassignment is carried out only when: (a) the number of active connections of the current base exceeds the prescribed threshold, AC, and (b) there is a server which is substantially less loaded than the current base. In the simulation experiments, described in the next section, the base is reassigned only when the least loaded server has fewer than AC/2 active connections.

Discussion We mention that in devising TDA we have been heavily influenced by the LARD (Locality-Aware Request Distribution) policy [10]. The goal of LARD is to assign HTTP requests so that access to disk — an expensive operation — is kept low. To this end, the dispatcher maintains for each document a set of back-end servers that have the document cached with high probability, and assigns a request for a document to the least loaded server from this set. There are, however, several important differences between LARD and TDA, and these stem from the type of the underlying applications:

- LARD treats all requests similarly in the sense that it assigns a request for a document to the least loaded base in the set. In contrast, TDA treats requests governed by stateless and stateful contracts differently.

- Under LARD a document can have several bases, whereas under TDA a stateful contract has to have a unique base; moreover, under LARD each document has a base, whereas under TDA only stateful contracts need to have a base.

We conclude this discussion by pointing out that TDA is more amenable to practical implementation than LARD. This is so, because a Web cluster may serve a very large number of distinct documents, and consequently the table storing the mapping between documents and bases can grow quite large. It follows that maintaining and consulting this table can be expensive operations. In contrast, contracts are substantially less numerous than documents, and thus, the corresponding mapping table is much smaller. Therefor, we expect that mapping contracts to bases is not onerous.

5 Experimental Performance Study

We conducted an experimental study to compare TDA with the LC (Least Connected) policy, which calls for the dispatcher to assign a job to the server with the least number of open connections. The policies were compared with respect to the performance metric of *waiting time*, computed as the time interval starting at request arrival in the dispatcher and ending at the point when processing commences (for a stateful contract, this point is reached after the contract state becomes available).

The simulation model consisted of a cluster of four back-end servers, and 100 contracts, of which 80% were stateful. Since no traces from e-commerce sites are available, the study used one of the data traces of request arrivals at a Web cluster, available on the Internet Traffic Archive[3]. Figure 3(a) depicts this trace, which contains approximately 170,000 requests over 200 seconds. The contract governing each of the requests was randomly chosen from the aforementioned set of 100 contracts.

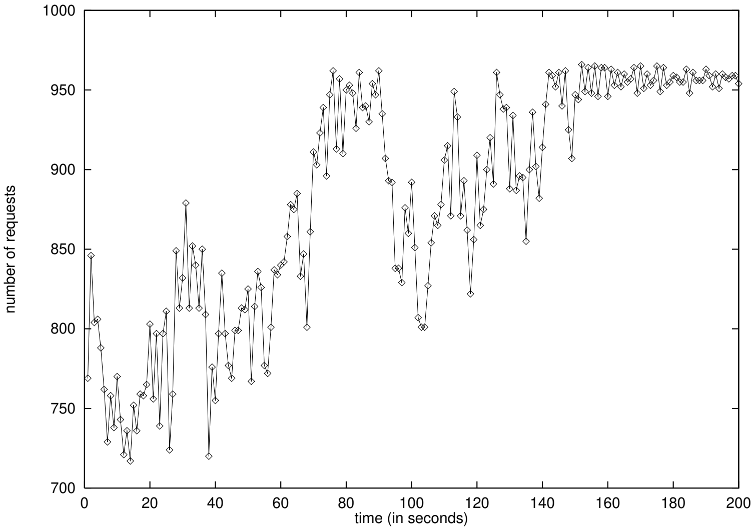
The upshot of the experimental results is that TDA outperforms LC by a factor of four. More specifically, the waiting time averages computed over the simulated time interval trace are 47.1 ms for TDA and 215.8 ms for LC.

We now proceed to detail the simulation study and the associated results. Standard LC has been extended in the simulation to ensure state consistency. Specifically, a request, R, governed by a stateful contract, C, is handled in the following manner. The dispatcher sends to the back-end server both R and the base of C. When the back-end server receives R, it requests the state from the base. After validating R, the back-end server sends to the base the updated state. The service time of a request is estimated as the sum of the time to verify compliance with the governing contract and the time to establish and close a connection. Furthermore, the simulation makes the following assumptions:

1. Communication times between the dispatcher and back-end servers are negligible.
2. The overhead incurred by the dispatcher to select (job, server) pairs is negligible;
3. The back-end servers are multi-threaded, where each thread performs all steps associated with a request before accepting a new one. When a thread blocks (because, for example, the state is used by another thread/server), another thread is chosen to run.

The parameters used by the simulation have the following values: (1) the time to evaluate a contract was measured empirically (see Section 3) as 3.3 ms; (2) the time to establish and close a connection was measured empirically (see [15]) as 0.9 ms; (3) the time to retrieve the state from another server is was estimated as 0.9 ms; (4) the threshold, AC was set at 20.

[3] http://ita.ee.lbl.gov/html/traces.html



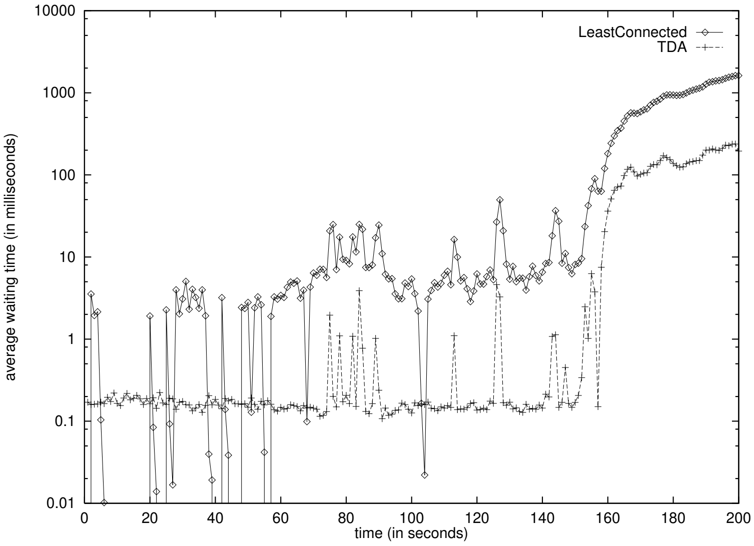


Figure 3: (a) Number of request arrivals; (b) Average waiting time of LC and TDA as function of time (notice that the y axis is logarithmic)

Figure 3(b) depicts the average waiting times in successive 1 second intervals. The figure shows that TDA improves over LC in almost all time intervals. A few exceptions occur only when the traffic is low, namely, when the number of incoming requests is less then 800 requests/second.

6 Related Work

There has been a growing interest in supporting e-commerce contracts, and a variety of different, and quite powerful, enforcement mechanisms have been devised. However, to the best of our knowledge, none of the frameworks proposed so far considered applying cluster-based architectures to e-commerce applications, nor to any applications governed by stateful policies. We will briefly review here those mechanisms that address specification and enforcement of contracts and discuss their scalability.

A number of researchers proposed that contracts should be expressed in a formal language, and should be enforced by generic trusted agents. This approach have been taken, in particular by Roscheisen and Winograd [13], and more recently by Dan et al. [4]. In both approaches, each contract is enforced by a dedicated trusted agent which internalizes the terms of the contract, and which is maintained for the duration of the contract. But having an agent for each contract can easily become a bottleneck, if the number of transactions between the enterprises involved in the contract is large. Furthermore, the number of trusted agents needed grows *linearly* with the number of contracts and might hurt overall performance[4].

Abiteboul, Vianu, Fordham and Yesha [1] implement contracts using *transducers*, which extend traditional relational database with rules, written in Datalog, stating the effects of an agent action (in this model, the observable events, which trigger policy evaluation, are database modifications). However, the mechanism deals with *only one* contract, and it is not clear whether several contracts can be supported in a scalable manner.

Finally, Ungureanu and Minsky [14] proposed a framework, called LGI, to support B2B e-commerce contracts. Enforcement is carried out under LGI, by associating with each communicating party, a trusted agent. The locality of enforcement solves the scalability problem: neither the number of participants in e-commerce transactions, nor the number of contracts is an issue any longer. However, scalability comes with a price: *all* parties involved have to use the same mechanism; it is still an open question whether LGI will be adopted and employed on such a large scale.

7 Conclusion

In this paper we propose an assignment policy for e-commerce clusters, called `TDA`, which takes into consideration request validation. We have shown experimentally that this approach results in good performance. More specifically, `TDA` outperforms performs `LC` by a factor of four (both policies consider the load of back-end servers, estimated by the number of active connections, when assigning requests).

However, this policy is probably not efficacious for every workload pattern. For example, if the workload is light, there is a high probability that whenever a request arrives there is an idle server on which it can be served without delay. In this case, traditional policies, like `LC` or Round-Robin, may perform comparably to `TDA` or even better. A real challenge is to find a way to continually adapt the dispatcher policy to changing workload patterns.

[4] Roscheisen notes in [12] that in this approach *"the number of objects ends up being quite large in realistic applications, and this quickly becomes infeasible, especially given the resources that each such object requires."*

References

[1] S. Abiteboul, V. Vianu, B. Forham, and Y. Yesha. Relational transducers for electronic commerce. In *Symposium on Principles of Database Systems*, pages 179–187, June 1998.

[2] V. Cardellini, M. Colajanni, and P. S. Yu. Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3):28–39, 1999.

[3] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.

[4] A. Dan, D.M. Dias, R. Kearny, T.C Lau, T. N. Nguyen, F. N. Parr, M. W. Sachs, and H. H. Shaickh. Business-to-business integration with tpaML asnd a business-to-business protocol framework. *IBM Systems Journal*, 40(1):68–90, 2001.

[5] Economist. E-commerce (a survey). pages 6–54. (The February 26th 2000 issue).

[6] Economist. Riding the storm. pages 63–64. (November 6th 1999 issue).

[7] Ron Lavi and Amnon Barak. The home model and competitive algorithms for load balancing in a computing cluster. In *The 21st Intl. Conf. on Distributed Computing Systems (ICDCS'01)*, April 2001.

[8] N.H. Minsky and V. Ungureanu. A mechanism for establishing policies for electronic commerce. In *The 18th International Conference on Distributed Computing Systems (ICDCS)*, pages 322–331, Amsterdam, The Netherlands, May 1998.

[9] N.H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *TOSEM, ACM Transactions on Software Engineering and Methodology*, 9(3):273–305, July 2000.

[10] V.S. Pai, M. Aron, G. Banga, M. Svendsen, P. Druschel, W. Zwaenepoel, and E. Nahum. Locality-aware request distribution in cluster-based network servers. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, 1998.

[11] V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[12] M. Roscheisen. *A Network-Centric Design for Relationship Based Rights Management*. PhD thesis, Stanford University, 1997.

[13] M. Roscheisen and T. Winograd. A communication agreement framework for access/action control. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, California, May 1996.

[14] V. Ungureanu and N.H. Minsky. Establishing business rules for inter-enterprise electronic commerce. In *Proc. of the 14th International Symposium on DIStributed Computing (DISC 2000); Toledo, Spain; LNCS 1914*, pages 179–193, October 2000.

[15] Teo Y.M. and R. Ayani. Comparison of load balancing strategies on cluster-based web servers. *Simulation, The Journal of the Society for Modeling and Simulation International*, 77(5-6):185–195, November-December 2001.