Dynamic Group Membership and Quiescence in a Supply Chain

Daniel Peek, L Julian Schvartzman, Yevgeniy Vorobeychik
Electrical Engineering and Computer Science
University of Michigan
{dpeek, lschvart, yvorobey}@eecs.umich.edu

12 March, 2003

Abstract

Supply Chain management has been a widely researched topic in Business, as well as Computer Science. While there has been a tremendous amount of theoretical work on the subject, few research implementations exist that explore issues such as market organization and quiescence detection within the supply chain. Our goal is to fill this gap. Consequently, we have implemented a supply chain that uses a group level abstraction to manage the markets for input and output goods at each level of the supply chain. Goods are sold at auctions managed by Market Servers. Auctions clear only when quescence is detected in the entire system, ensuring the designer's control over global efficiency of the system. Autonomous agents were implemented to take advantage of this framework and simulate behavior of actual economic agents participating in a supply chain. In general, though, any agent can be designed to participate in our supply chain system, as long as it adheres to the interface specifications. We also introduce a complication to the common theoretic model of supply chains by dropping the assumption that agents will be correct (i.e. non-buggy). Our system can handle crash failures of agents through the use of well-understood group membership semantics. Finally, as a strong incentive for agents to behave appropriately (and for designers to eliminate agent errors that would negatively impact other agents), we introduce a reputation system, in which an agent's reputation score affects the probability of that agent's participation in a given round of transactions.

1 Introduction

Distributed electronic auctions are becoming increasingly popular in e-Commerce, and much research activity has been devoted to organizing autonomous agents into electronic markets to increase efficiency and reduce transaction costs. Several important problems inherent to many distributed systems are relavant here: termination (quiescence) detection, dynamic membership, and performance and failure control. Quiescence detection has been explored by Dijsktra and Scholten [3], Mattern [5], and others in the context of generic distributed systems, as well as Wellman and Walsh [9] in the context of supply chain management, which is the area of particular interest to us. Dynamic membership is another age-old problem in distributed systems, and is especially difficult when the system relies on quiescence detection to end a period of activity, since agents can enter and leave the auction at any point without notifying the servers. Finally, we implemented the reputation system that serves to reduce the incentives of individual agents to cheat, and thereby address the question of performance and failure control.

As already mentioned, we will develop a supply chain that relies on central auction servers to "run" each respective market along the supply chain. Auction servers are assumed to be reliable and to have reliable links between them. At the beginning of the quiescence detection protocol, an activator process propagates activation through the supply chain network. After that, termination detection proceeds on two levels: the inter-group and intra-group level. On

the inter-group level, termination detection process is very similar to Dijsktra and Scholten, while within groups it primarily uses time outs and agent activation counters to detect when agents become passive. An important additional assumption imposed here is that markets will eventually terminate, since at some point global equilibrium will be reached and no agent will have an incentive to continue bidding. Furthermore, to prevent auctions from lasting indefinitely, we will impose a minimum bid increment. Otherwise, agents can bid infinitesimally small increments above previously winning bids.

The dynamic membership problem complicates termination detection, as members can enter and leave any market at any point during the auction. If an agent leaves the market after having submitted a winning bid, a quiescent state may be incorrectly assumed by the activator. Instead, bids of non-existent agents must be automatically "decommitted" – withdrawn from the markets – as other agents will probably change their bidding behavior based on this additional information. Decommitment (voluntary or automatic) may also result in a penalty imposed on the agent, which is reflected in the agent's reputation score and effectively reduces the agents expected future payoffs. The agents may also incur a pecuniary penalty for decommitments. In our current implementation, we impose no decommitment penalty, but do reduce the agents' reputation scores when they decommit by failing.

The reputation system works through authentication. When an agent enters the group (any group along the supply chain), it is authenticated by the group leader. The purpose of authentication and the reputation system is to discourage the agents from "misbehaving" by entering spurious bids or deviating from the bidding and quiescence protocol. The reputation system may also serve to detect faulty (buggy) agents and exclude them from the market after an excessive number of violations.

Babaioff and Nisan [1] suggest using double-auctions across the supply chain, and this is the road we will take. When quiescence is detected, all sell bids are matched against the corresponding winning buy bids and the markets clear at a price specified by the auction rule. At this point, there must not be any lingering bids from non-existing agents, as those would have been cleaned up in the process of termination and dynamic membership detection mechanisms. After "cleaning up" all the markets, the activator initiates the next period of exchange by activating all auction servers again.

2 Design and Implementation

2.1 Groups and Group Membership

2.1.1 Preliminaries

A typical supply chain consists of multiple markets at which input goods are purchased and output goods sold by the agents. All transactions between agents are centralized, with market servers mediating these transactions using an auction mechanism. We assume for convenience that each market server mediates a sale of a single good from some supplier agents to some other consumer agents. Figure 1 shows an example of a supply chain as we envision it.

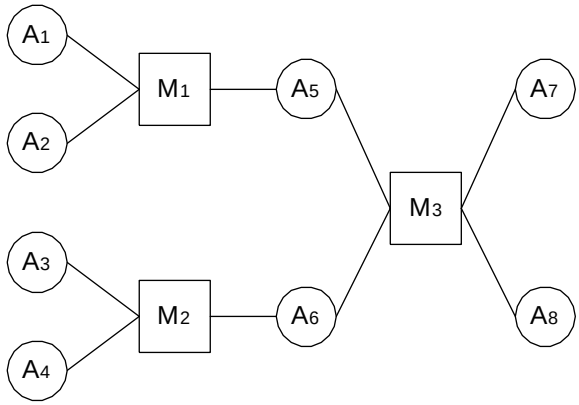

Figure 1: An example of a supply chain

As the number of market servers and agents grows, the need arises to separate the supply chain into a set of linked components, or groups, in the spirit of the divide-and-conquer approach. Such treatment allows us to view each group as a monolythic component, and globally be concerned primarily about inter-group communication and some minimal global state. Naturally, this will only reduce complexity if

we can make some assumptions that limit interdependence between members of different groups. Thus, before proceeding with a more detailed discussion of our group membership and quiescence detection protocols, we will explicitly state our assumptions and observations that we can make given these assumptions. Also, to clarify our logical framework, we present the definitions for supply and demand groups that will be used in the remainder of the paper.

Definition 1 *A* supply group *of a market server includes the server itself, all the agents that are selling goods at that server, and all the market servers through which these agents are buying goods.*

Definition 2 *A* demand group *of a market server includes the server itself, all the agents that are buying goods through that server, all the other market servers through which these agents are buying inputs, and all the market servers through which these same agents are selling goods their goods.*

It is apparent from the above definitions that a particular group is a Supply or Demand group from the perspective of some market server, and can thus be referred to as simply a *group* if further specification is of no consequence. Thus, since each agent belongs to some group that is a supply group from the perspective of the market server through which that agents sells its output good, and a demand group from the perspective of the market server through which it buys its inputs. Figure 2 shows how groups in a supply chain are formed based on our definitions.

Having defined groups, we can now use them to abstract away the details of the supply chain in Figure 2 by viewing it as a graph of groups, as in Figure 3.

In order to simplify our design and analysis, we made the following assumptions to restrict our model of the supply chain:

Assumption 1 *Agents in the same group can only sell their goods through exactly one market server.*

Fortunately, with this assumption we lose no generality, since an agent that wants to sell multiple products can simply acquire multiple agent profiles at registration.

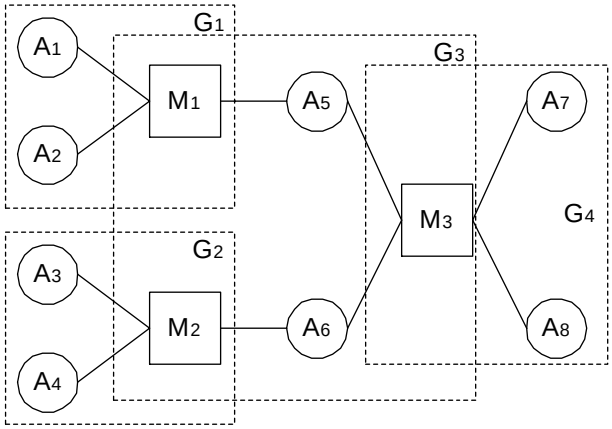

Figure 2: Formation of groups in a supply chain

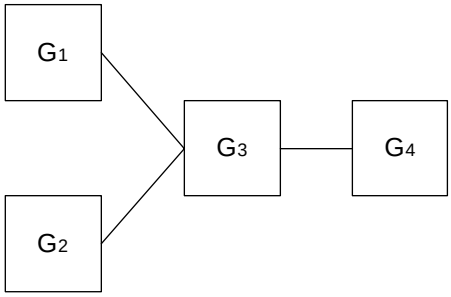

Figure 3: Viewing supply chain as a graph of groups

Assumption 2 *Agents that sell their goods through the same market server must also buy their inputs from the same set of servers.*

While this assumption limits generality, we believe that it is valid for a large number of real supply chains. Still, in the future we hope to relax it to gain generality.

Assumption 3 *Agents that buy their goods from the same set of market servers must also sell their goods through the same market server.*

While this assumption also greatly limits generality, it ensures that the market servers in our supply chain form a tree. While we do not believe this necessarily to be a typical case, we do think it is common enough to deserve special attention. As with other restrictive

assumptions, however, we will attempt to remove it in the future.

To reiterate an assumption stated slightly earlier,

Assumption 4 *Each market server mediates a sale of exactly one good.*

This assumption is here only for convenience of exposition, as we can easily envision a market server that mediates multiple auctions as multiple logical market servers.

Armed with these assumptions, we can now make several observations that help us with the design of our group membership protocol.

Observation 1 *Each market server belongs to exactly two groups: a supply group and a demand group. Both are formed from its perspective according to the Definitions 1 and 2.*

Proof: It follows directly from Definitions 1 and 2 that each market server will form a supply and a demand group. Thus, we only need to show the following for some market server M_1: first, if some other market server, M_2, forms a demand group which includes M_1, this group is identical to the supply group that M_1 itself forms; and second, if some other market server, M_3, forms a supply group which includes M_1, this group is identical to the demand group that M_1 forms.

Let us start with the first part. The members of the demand group formed by M_2 are the set of agents, $\mathcal{A}$, that are buying goods from M_2, the set of all market servers, $\mathcal{M}_{supply}$, through which these agents are buying goods (note that this set includes M_2), and the set of all market servers, $\mathcal{M}_{demand}$, through which agents in $\mathcal{A}$ are selling their outputs. From Assumptions 1 and 3 it follows that $\mathcal{M}_{demand}$ contains exactly one market server, which is M_1, while Assumption 2 ensures that the set $\mathcal{M}_{supply}$ is the same as the set of market servers that are members of M_1's supply group.

The argument for the second part is symmetric to the one just presented, with M_1 now serving the role of M_2 above, and M_3 forming the identical supply group.□

Observation 2 *No two market servers can form identical supply groups. In fact, market servers must form unique supply groups.*

Proof: We can prove this by contradiction. Let's suppose that some two market servers, M_1 and M_2 form identical supply groups. The composition of both of their supply groups is a set of agents, $\mathcal{A}$, which sell through both of these servers, and a set of market servers, $\mathcal{M}_{supply}$, through which agents in $\mathcal{A}$ are buying goods. Clearly, agents that buy their goods through the same set of market servers $\mathcal{M}_{supply}$ are selling through both M_1 and M_2, which is a violation of Assumption 1.□

Observation 3 *No two market servers belong to the same two groups.*

Proof: Let's suppose that some market server M_1 forms the same supply and demand groups as some other market server M_2 (any other way for two markets to belong to the same two groups would violate Observation 1). It follows from Observation 2 that M_1 and M_2 must form unique supply groups.□

Observation 4 *Each group contains at least one market server.*

Proof: This follows directly from the difinition of a group, since a group is formed from the perspective of a market server (if there were no market server in a group, no group would have been formed).□

Observation 5 *Each agent is a member of exactly one group.*

Proof: Let's say that some agent, A_1, belongs to a supply group G formed by some market server, M_1. From Observation 2 we know that it can belong to only this supply group. Let $\mathcal{M}_{supply}$ be the set of market servers through which A_1 buys its inputs. Since all agents in M_1's supply group must buy their goods through $\mathcal{M}_{supply}$ and sell through M_1, we can suppose without loss of generality that A_1 is the only agent in this supply group (it can be some meta-agent that represents all the agents in this group). Since M_1 forms exactly one supply group, and each of the market servers in $\mathcal{M}_{supply}$ forms exactly one

demand group which is identical to G, G must be the only group to which A_1 belongs.□

Since each market server uniquely belongs to some two groups, it can serve as the medium of communication between these groups. This would be a moot point if there was a source of interrelationship between the groups other than a particular market, but to combat this difficulty, we will make another independence assumption:

Assumption 5 *The only source of interrelation between any two groups is through the state of the market server that belongs to both.*

While this assumption limits the communication between agents and market servers, it ensures, along with the assumptions 2 and 3 (agents that buy/sell their goods in the same set of markets must buy/sell their goods in the same set of markets as well), that our supply chain groups form a tree with links representing interrelations between groups. As was true of several other assumptions, this one is made to ensure tractability of protocols we devise with this model, and in the future we will look to gain generality by eliminating assumptions.

Finally, our most heroic assumption:

Assumption 6 *Market servers are reliable and trusted.*

Actually, we do not believe that the first part of this assumption is particularly constraining, as there are numerous methods to ensure reliability of each individual market server, the most obvious being replication. Recall that any market server is just a logical component, and can internally be arbitrarily complex. While replication will not ensure 100% reliability, we think probability of failure can be made small enough to be of no concern.

Trust is slightly more worrisome, as we can envision malicious servers on the network masking as members (possibly, by compromising one of the trusted market servers). For now, the problem of trust in this context is beyond the scope of this work, although we will try to address it in the future.

2.1.2 Group Membership Protocol

Since we had already made the assumption that our market servers are reliable, we can assign some market server in each group to be that group's leader without concern about centralizing the point of failure. Note that if we actually allowed any market server to fail, our supply chain would have no hope of functioning, since each market server represents a unique and essential good. This makes reliability of individual servers orthagonal to our discussion (though no less important). If a group has only one market server, this server becomes the group leader. If, however, there are multiple market servers in a group, the leader of this group is the server that views this group as its Supply group.

Observation 6 *There is exactly one group leader in each group.*

Proof: Certainly, this will be true for groups with only one market server. In a group with multiple market servers, only one will view it as its Supply group. To show this, suppose that (without loss of generality) two servers view this group as their Supply group. Agents that sell to through the first server will purchase from a different set of market servers than agents that sell through the second, per observation that agents buying from the same set of servers must also sell through the same market server. But then their is no direct interrelation between these two servers and agents that are selling through them, and so they must belong to different groups.□

An agent can join a group at any time by authenticating with the group leader – a process which is described in more detail in the Authentication section. Once authenticated, it is added to the group leader's member list and can now sell goods at some market server in that group and purchase input goods from others. It does not need to communicate with any other agents, since all transactions are mediated by the market servers.

Agents can also leave the group at any point in time. If they have any winning bids outstanding at the auctions, they must explicitly decommit (in other words, withdraw their bids). However, it is possible that agents will leave without decommitting (for

example, because of a crash or network partitioning), potentially causing quiescence to be detected prematurely. To guard against this possibility, the group leader will exchange heartbeat messages with all agents. Thus, an agent's disappearance will be discovered within a bounded time period. Once the leader concludes that some agent is no longer a member of the group, that agent is removed from the membership list and all its bids are automatically decommitted. Any further messages from that agents are subsequently ignored until it explicitly rejoins the group. Upon rejoining, the agent will find that it no longer has any outstanding bids, and is completely responsible for making its state consistent with current group state and resubmitting bids.

2.2 Quiescence Detection Mechanism

Our quiescence detection protocol takes advantage of the abstraction layer provided by the groups by separating the tasks of global (inter-group) detection and local (intra-group) detection. Global detection is handled by an entity we will refer to as an *activator*, which initiates a period of transactions by sending a "New Auction" (activation) message to a group with "consumer" agents – agents that purchase good for final consumption[1] Once a group is activated, the group leader activates (by sending the "New Auction" message) all the groups that are providing inputs to agents in this group. Since this pass of the protocol only stops when there are no more groups that can be activated by any already active group, all relevant groups will eventually be activated. A group is considered relevant if there exists a path from the activator to that group. Otherwise, a group is irrelevant. Note that since the interconnections between groups form a tree in our model that is rooted at the activator, a group is relevant if and only if it is a node in this tree.

Once a group is activated, it activation within it is trivial: the group leader simply broadcasts the "New Auction" message to all group members, agents and market servers alike. As we had mentioned earlier, every market server activated in such a way belongs uniquely to some two groups. Additionally, by the definition of a group leader such a market server must be mediating some input market (in other words, it views this group as its Demand group).[2] Given that each input good will be an output good in some group the market server mediating its sale will be the group leader in the group in which this good is sold. What follows from this sequence of observations is that the group activation process is a result of local activation, with market servers providing the connections between groups.

Now that we have demistified the process of inter- and intra-group activation, we must explain how the subsequent detection and finalization (clearing of all markets after quiescence was detected) takes place. The process proceeds as follows. In general agents are passive. However, once an agent is activated, it is allowed to submit at most one bid at each of the markets in its group, after which it becomes passive, after which it is unable to bid again (any subsequent bid requests are ignored, until the agent is explicitly activated). Agents also become passive by explicitly notifying the market server or by taking no action within a fixed amount of time (the timeout period). Agents can be reactivated (and therefore made able to submit bids again) if there is a change in the top bid of any market that agent is connected to. This change may occur because some active agent had submitted a winning bid, decommitted a winning bid, or had dropped out of the group (explicitly or implicitly). Market server that detects some change in its state notifies the leader of its demand group of this change, and all market servers at its supply group. Subsequently, it activates all agents in its supply group, while the group leader of its demand group first sends State Change Notification to all the market servers in that group, and then activates all the agents in the group. Since we are assuming reliable point-to-point communication, we are confident that when the reactivated agents submit bids, these bids will be expected. Finally, to account for the possibility of multiple nearly concurrent state changes, we allow agents to bid multiple times at each market server, but no more than the number of activa-

[1] More formally, this group cannot be a supply group for any market server.

[2] Recall that unless it is the only market server in the group, a group leader must view this group as its Supply group.

tions. Market servers accomplish this by keeping a counter for each active channel. When the value of the counter reaches zero, the corresponding channel (and, therefore, agent) is considered passive.

Eventually all markets will consider all agents to be passive. Rational agents will reach a maximum price they are willing to pay and stop bidding. Broken agents may bid prices higher and higher if buying or lower and lower if selling, but this is limited by a maximum price and the inability to sell for less than zero price. Conversely, broken agents may never communicate at all, but would then be considered quiescent after the timeout period has elapsed.

Markets are considered quiescent when each agent attached to that market is quiescent. To detect the situation in which every market is quiescent, the activator periodically sends out quiescence queries to the top level markets. If those markets are quiescent, they send a similar query to each of their input markets. If those markets are quiescent, they send the query on to their input markets and so on. If the query reaches a market which is not quiescent, a negative response is returned. If the query reaches a quiescent market without input markets, a positive response is returned. Each market collects responses to the query and if any are negative responses, returns a negative response. Finally, the accumulated response is returned to the activator. If the response is negative, it knows that the system is not yet quiescent. If the response is positive, then the system is quiescent. The activator then notifies the markets that the system is quiescent and that they may clear their auctions. Then activation messages are sent out again and the whole process repeats.

There is an unfortunate case in which the quiescence detection process can result in a false positive. If an agent with the top bid disconnects after an affirmative response to a quiescence query has been sent, one or more markets may not be quiescent, but the activator may still receive an affirmative response. To reduce the likelihood of this event, each market can send a message directly to the activator to request that any outstanding quiescence queries are considered to have negative responses. However, there is still a possibility that quiescence is detected falsely. For example, an agent can fail precisely when quiescence is detected by the activator, and since there is non-zero message transmission delay between any market server and the activator, the activator will not "catch" the problem in time. Note that waiting a finite time period after detecting quiescence does not solve this problem! The reason that this situation cannot be completely eliminated is that solving this problem is equivalent to solving the distributed agreement problem and our system does not have sufficient communication capabilities to solve it.

2.3 Authentication

In order to join a group, the agent contacts the appropriate Group Leader with an authentication request. The Group Leader validates the agent according to its local database. If the validation is unsuccessful, the error message is returned to the agent. Otherwise, the server caches a unique agent id, adds the agent to its group membership list, and responds with a positive acknowledgement. Agent id is used as a key into the agent reputation database.

There are several purposes of authentication in our system. One is allowing an agent to identify which group it wants to join, in order to maintain the group level abstraction we had started with. Another is to support our reputation system, which identifies an authenticated agent by its name. Without some form of security, both of these are moot, and it was our goal at first to implement a public key/private key system to ensure authenticity of agent messages. What we did not realize at the time is that CAEN machines have a relatively old version of Java VM, and the Java library that facilitates implementation of cryptography is more recent. Since we had to run multiple markets, agents, and an activator on separate machines (all agents can run on the same machine, however), we had to use the computing facilities of CAEN to attain any meaningful results. Thus, security is not a part of the current version of our system.

2.4 Reputation System

In order to police the agents and ensure reasonable behavior in the system, we implemented a reputation system. Our first requirement from the reputation system was that it would allow agents to recover from

bad reputation in the long term. Second requirement was that agents with no reputation score have an incentive to improve their reputation (in other words, there is sufficient disincentive to behaving badly and then starting back at zero with no penalty; agents can rid themselves of bad reputations quite easily in general, since they can simply register a new profile, as is the case with eBay).

To satisfy the above constraints, we decided to use reputation score as a determinant of probability of an agent participating in a period of transactions. We assume, of course, that the probability of participating in a period of supply chain transactions is directly proportional to agent's expected future payoffs. This seems reasonable, since an agent will have some expected payoff for each game, which would be multiplied by probability of participating in a game. Additionally, this also excludes misbehaving agents from the system (in the limit, since probability approaches zero with increasingly low reputation score), which is a convenient side-effect.

Now that we know that probability of an agent participating in a supply chain at any given period is a function of its reputation score, we need to specify precisely what this function is. We chose it to be an S-curve with the lower bound of zero and the upper bound of one. This suits our purpose quite nicely, since we are bounded by the minimum and maximum probability values, and, as we would expect, negative reputation score causes the probability of participation to approach zero in the limit, while the probability approaches one as the reputation score rises. The point of inflection is at the reputation score of zero: this is the probability with which an agent starts out. We can vary this as a part of the system design to impose a certain type of behavior on the system based on our expectation of agent behavior and its effects on the system. The particular function that we chose is

$$p(r) = \frac{1}{1 + (\frac{1}{k} - 1)e^{-B \cdot r}}$$

where $p(r)$ is the probability of participation as a function of reputation, k is the probability of participation of an agent with zero reputation, and B adjusts the speed of convergence to zero or one with reputation score, and will depend on the behavior that we want from the system, as well as the dimensionality of reputation scores.

In our current implementation, reputation is adjusted in the following cases:

- When the market clears and a buy bid matches a sell bid; senders of those bids receive a boost to their reputation scores.
- When the market clears; all agents that participated in the auction receive a reputation score boost.
- When an agent decommits (e.g. by disconnecting explicitly or through crashing); its bid is erased from the auction server and the agent's reputation score is lowered.

Before the markets open, each group leader has an opportunity to prevent agents in their group from participating. Group Leader of each group computes the participation probability for each agent and then "flips the coin" to decide whether each agent participates in the next period of exchange. After determining the group membership for the next period as a result of the coin flip and agents simply entering and leaving, the Group Leader sends a message to each supply market to indicate the group view change. Banned agents are not activated when the markets open or when changes are made in related markets. Bids received from banned agents are ignored. The ban on participation expires when quiescence is detected. At that point, the group leader can decide whether to renew the ban.

2.5 Market Server Design

At the application level, Market Server runs an auction to mediate the sale of a unique good.

At the application level, a Market Server (or simply a Market) runs an auction to mediate the sale of a unique good. The auction is characterized as *Double-sided* since it admits bidding from both buyers and sellers.

A Market is said to be *open* when it is activated, and it remains in that status until quiescence is detected. While open, a Market is responsible for receiving bids and enforcing bidding rules, erasing bids

of those agents that decommit (i.e. agents that disconnect), updating the state of the auction (which includes bookkeeping for all the bids received), revealing information to agents, and updating the reputation of each agent (as described in 2.4). Upon quiescence notification the Market clears, matching buy offers with sell offers and notifying transactions appropriately.

Each bid is a price-quantity pair. For simplicity, we did not implement bids with multiple price-quantity points. To enforce a discrete good rule, only integer quantities are accepted. In addition, prices are restricted by a minimum possible amount change *MPC*, and agents are required to submit bids that dominate their previous bid. A buy bid for a price p_{new} is said to dominate a previous bid for a price p_{old} if $p_{new} \geq p_{old} + MPC$. Similarly, a sell bid dominates a previous bid if $p_{new} \leq p_{old}$ - *MPC*. Note that an agent could submit a bid that does not dominate its previous one by decommitting and later bidding again. This behavior, however, is discouraged, and the reputation system penalizes it.

Information revealed to agents during an auction consist of the standard price quote, which is the bid-ask spread. *Bid* is the maximum price that any agent is willing to pay for a good. *Ask* is the minimum price that any agent is requiring to sell a good. This bid-ask quote is pushed to agents as soon as the quote changes, which occurs upon receiving a new highest buy bid or a new lowest sell bid, or upon agents decommitting the highest buy or lowest sell bids.

When quiescence is detected, the Market clears and executes a simple allocation policy that determines which agents transact, at what price and for what quantity. The algorithm proceeds as follows. Bids are sorted by price, and highest buy bids are matched to lowest sell bids. If the *buy* price is greater than or equal to the *sell* price, they transact for a single unit and the owners of the bids are informed. The price of the transaction is the average between the *buy* and *sell* prices. The algorithm loops and evaluates bids unit by unit until no more units are offered or demanded, or until no match is found because the buy offer is lower than the sell offer. Ties between bids for the same price are broken arbitrarily.

A more general and parameterizable auction server can be found in The Michigan Internet AuctionBot [11]. We decided not to use AuctionBot for our purpose, since it is more complicated that our purpose requires, and would need to be modified to interface with our middleware. Thus, it was easier to write our own auction server instead of modifying AuctionBot.

2.6 Agent Design

Each agent in our supply chain is an instantiation of the agent class, and can follow one of a set of potential generic strategies based on the parameters that are set as a part of object instantiation. We will rely on the extensive body of strategic agent literature and Trading Agent Competition experience (e.g. [2]) to design reasonable agent strategies in this environment. Further, we will assume that agent's preferences are exogenous to the system. Based on these preferences, each agent bids for supplies in the supply markets by sending its bids to the appropriate supply-side auction servers until the prices for goods exceed its value for them. Similarly, it attempts to sell its outputs at the demand-side auction. Current implementation of the agent does not withdraw bids from auctions. We will now describe the agent implementation and strategy in greater detail.

Implementation Overview The agent is implemented as a subclass of Rational Agent. Rational Agent is an abstract class that encapsulates much of common agent functionality in our system and implements the Java interface Agent. The extension of Rational Agent is then responsible for "filling in the gaps", i.e. providing an initial and interim bid function, enforcing bidding rules, implementing the drop out predicate, etc. We developed three implementations of the actual agent subclass:

- *Competitive Agent*: a generic agent, which bids for inputs and sells some output
- *Consumer Agent*: the agent that perform the final consumption in the supply chain, i.e. does not produce any goods. The values of input goods to this agent are derived from its utility, which is exogenous.

- *Producer Agent*: the agent that provides initial production inputs. This agent already has the input goods and either resells them or resells the immediate product from them. Its input prices are exogenous prices of procuring the raw materials or producing the immediate output good.

Agent Parameters Since our goal was to make the agent flexible to facilitate future development, the agent was parametrized. Several of the obvious parameters are fixed cost and opportunity cost. Another is decommitment cost, which is currently simply added to the value computation. This and other system-wide (or group-wide) parameters will in the future be supplied by the system. Finally, auction rules and internal bidding rules are provided as parameters. These may be either system-wide or exogenous parameters (e.g. the agent may wish to enforce rules beyond auction requirements).

Price Prediction In order to determine what amount to bid for a particular good, the agent needs to estimate its value for that good. Since a rational agents tries to maximize expected future value, it is typical to estimate value for a particular good using predicted price of that good. In general, price prediction is not trivial, as can be seen from the design of trading agents in the Trading Agent Competition [8]. The most common approach is using some form of a historical average, and since it is also the simplest to implement, this is the approach we took. Our initial price prediction for the price of the good is arbitrary, and we do not use it in our moving average calculations. It may be worth noting, however, that since all agents that we deploy use essentially the same strategy, it is likely that the final price will be not far from the predicted price.

Computing Value of Goods The value for a good is computed based on additional profit we collect by owning this good. For now, our production model involves only one unit of each good, and, consequently, value of each good is our total profit (i.e. we need exactly one unit of each input good to produce the output good, and we need to sell the output good to make any money). Profit is computed as total revenue less total cost. Our cost is composed of fixed cost, opportunity cost, and variable cost components. Variable cost is per unit of each good.

Computing Bids Since the agent participates in an English auction which allows agents to bid incrementally, our buy bids start at the lowest possible value (in this case, zero), while our initial sell bids are some initial high value (for now, 2*value_of_good). We then incrementally raise the buy bids and lower sell bids, with the increment being a specified parameter (we use ten to speed up convergence).

We submit a computed bid if it is higher/lower than our previous bid, higher/lower than current quote, and below/above our value for the good. Once the current quote exceeds our value for a good, we stop bidding[3].

Bid Validation Here we use the auction and bidding rule parameters to validate the bids before submitting them to the auctions. If validation passes, bids are submitted, otherwise, the agent may either revise the bids or simply return.

Drop Out Predicate To allow agents to drop out of the supply chain, we implemented a drop out predicate, which is checked at the beginning of every period. If this predicate ever becomes true, the agent exits the supply chain. Currently, this is implemented at the Rational Agent level, but the derived agents do not use it.

3 Experiments and Results

In order to evaluate our system, we ran a series of experiments to determine how time to quiescence varies in the numbers of agents and markets. What we found, as we describe in the next several sections, is that the effect of an increase in agents and markets is minimal, and, thus, we can conclude from these initial results that our system is relatively scalable. This is not completely justified, since we ran at most 32

[3]Typically, if we at this point believe that we cannot get a complete bundle of inputs or cannot sell our output, we would decommit all of our bids. We had not implemented this, since it is a trivial detail and does not add any value to our work.

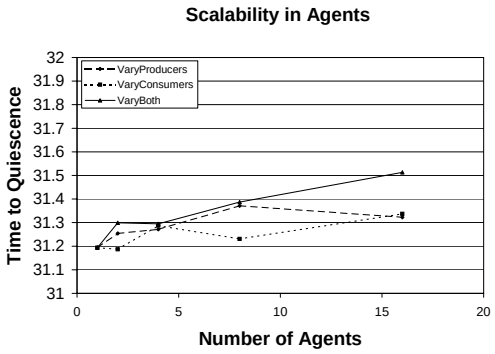

Figure 4: Time to quiescence dependence on number of agents

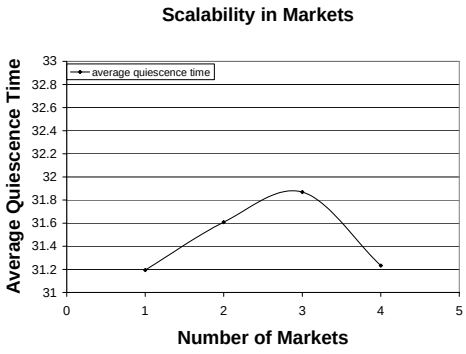

Figure 5: Time to quiescence dependence on number of markets

agents and 4 markets (1 market for each level in the supply chain), but the evaluation gives us an idea of the kind of an effect agents will have on performance. The experiments were run on University of Michigan CAEN computers, with activator and each market server running on a separate machine, and all agents running on the same machine.

3.1 Scalability in Agents

We ran a series of experiments to determine how our system scales with the number of agents. All of these experiments were run with 1 market server.

We varied the number of agents between 1 and 16, doubling the number of agents at each subsequent experiment. This was done for demand side agents, supply side agents, and then both. The results are presented in Figure 4. As can be seen from the graph, variations are insignificant, mostly due (we speculate) to variations in network congestion during the experimentation.

3.2 Scalability in Markets

To test how our system scales with the number of intermediate markets (i.e. levels in the supply chain), we ran similar experiments to those described in the previous section. The number of markets was varied between 1 and 4, all running on separate machines. The results are presented in Figure 5. We can see from the results that, just as in the case of agents, number of markets has little effect on time to termination. This is somewhat surprising, and it is likely that a considerably larger number of markets will have an impact. We had not at this point tested this hypothesis, but intend to create a more sophisticated supply chain in the future. Since it would be difficult to manually start up a large number of markets, and since agents will need to know the locations of the markets, and, finally, since all markets need to run on separate computers, it is a non-trivial task to set up a large scale experiment in terms of market servers.

3.3 Effects of Joining and Leaving

We only present informal results here, since the effects of a given agent dropping out of a supply chain are negligible. The only actual effect is that, since its bid is erased, a state change occurs at the market at which its bid was registered, though only if this was a winning bid. The semantics of state change notification are as described by our quiescence detection algorithm, and there are only insignificant transient effects. Having said this, an agent may possibly drop out as quiescence is nearly detected. Thus, we will have lost this detection period, which is at most 30 seconds, to a spike in activity. Effects of joining are even more minimal, since the new agent does not bid on any auction until it is notified of a new auction or a state change.

4 Conclusion

As one of very few research implementations of Supply Chain Management systems, our system attempts to address the issues of detecting termination and increasing reliability through an incentive system in a very specific supply chain structure. The structure we examine is a tree, though we are unsure whether the same basic framework will work if some of our assumptions are dropped. That is a topic of future research. We tested an idea of a probabilistic reputation system as a means to create incentives for agents to behave correctly (and for agent designers to create correctly behaving agents). After running some tests to determine scalability of our system in the numbers of agents and markets, we discovered that, as far as we can tell, there are few scalability problems. Obviously, problems may only arise after some critical number of agents or markets is exceeded. We have no intuitions to offer regarding such a case, and it remains to be seen what our scalability behavior is like in increasingly complex environments. For now, we have only begun validating some of our ideas, and have gained some invaluable experience about supply chain design and implementation.

5 Acknowledgements

We would like to thank Michael Wellman, Rick Wash, and others for their helpful comments and suggestions.

Appendix A: Programming Interface

Agent API

Upon program execution, the AgentInterface loads the class implementing the logic of the agent named as a command line argument and instantiates that class. During execution, it manages the quiescence state of the agent.

The interface exposed to the application level agent is referred to as Market Representation, since presumably the agent will be "talking" to some Market Server. This interface includes the following functionality:

- **GetQuote:** Asks the Market Server for the current price quote.
- **SubmitBid:** Submit a bid to Market Server. Returns bid state (whether bid was accepted or rejected).
- **WithdrawBid:** Decommit a bid from an auction. There is a cost for decommitting a bid, but for now we consider this an "offline" cost.
- **GetBidState:** Asks the Market Server for the current state of its bid.

Agent Interface

The agent at the application layer will expose the following interface to the middleware layer by implementing a Java interface Agent:

- **AuctionCompletionNotification:** Agent is notified that the auction has completed and receives the quantity it had won and the price it must pay.
- **StateChangeNotification:** This is the activation message, notifying the agent that it can now submit a bid to the auctions (it is active).
- **NewAuctionNotification:** Agent is notified that the new auction had begun. Equivalent to an activation message.

Market Server API

The Market Server interface first loads the class implementing the logic of the Market Server specified on the command line and creates an instance of that class. During continued operation it handles requests incoming requests and passes them on to the Market as necessary. It is also responsible for detecting the quiescence of the agents associated with it and passing that information onto the quiescence detector.

At this level, the middleware layer exposes an interface to communicate with market agents to the

application layer. We call this interface Agent Representation, as it is an abstraction of the agent with which the Market Server is communicating. This interface supports the following functionality:

- **AcceptBid:** Notifies the agent of acceptance of the bid submitted by that agent.
- **RejectBid:** Notifies the agent that its bid was rejected.
- **NotifyNewAuction:** Notifies the agent of the start of the new auction period.
- **NotifyStateChange:** This is essentially the activation message to the agent, as it tells the latter that the Market Server state had changed, implicitly asking the agent to react (possibly, by submitting a new bid). A price quote (current high bid) is sent as a part of this message.
- **ClearAuction:** Notifies the agent that the auction has cleared and sends this agent quantity and price of the good it won. This message is sent to all agents, and if a particular agent had not won any quantity of the item, the "quantity" field of the message is set to zero.
- **Ban:** Bans the specified agent(s) from participation in the next round of transactions. This is done by the Group Leader.

Market Server Interface

At the application layer, the following functional interface is exposed to the middleware layer by implementing a Java interface Market:

- **NewAuction:** Server resets its state and gets ready to receive new bids from agents.
- **BidNotification:** Server is notified of a new bid submitted by an agent. It updates its state accordingly and responds to the agent with the results of the call (i.e. bid accepted or rejected). If the state change requires activation of other agents, the server notifies them of such a change.
- **Decommit:** Server is notified of a bid withdrawal (decommitment request).
- **GetQuote:** Server returns current price quote to the requesting agent.
- **GetBidState:** Server returns its state, i.e. price (BID/ASK) quotes, and most recent bid of the agent issuing the call.
- **NotifyQuiescent:** Server is notified that quiescence was achieved in the whole supply chain. It calculates allocation of goods and notifies agents of the outcome.

Appendix B: Process Communication Details

Communication Between Agent and Market Server

Agent communicates with the Market Server through a TCP socket connection. Agent sends the following messages to the server:

- **Join:** Establish connection
- **Leave:** Disconnect
- **Bid:** Place a bid
- **Decommit:** Withdraw a bid
- **GetQuote:** Gets quote from the market

In return, the Market Server will send the following messages:

- **Join-ACK:** Accept a request to join
- **Join-NACK:** Reject a request to join
- **Bid-ACK:** Bid Accept a bid
- **Bid-NACK:** Reject a bid
- **NewAuction:** Notify about the start of a new auction
- **ClearAuction:** Notify the agent of its winnings
- **StateChange:** Notifies the agent of a change in the ongoing auction

Communication Between Agent and Group Leader

Besides talking to Market Servers, each agent will need to exchange some messages with the group leader of its respective group. Here are these messages:

- **Auth:** Agent asks the group leader to be added to the group and submits authentication information (i.e. username and password).
- **Auth-ACK:** Group Leader inform agent that authentication was successful
- **Auth-NACK:** Group Leader inform agent that authentication was unsuccessful

Communication Between Market Server and Group Leader

Finally, Market Server needs to exchange a few messages with the Group Leader as a part of our protocol:

- **Activate:** Group Leader (or, initially, the activator process) notifies Market Server that the new round (period) had begun.
- **Quiescence-Query:** Group Leader (or activator) asks the Market Server if the latter is quiescent.
- **AuthNotify:** Group Leader notifies the Market Server of a group membership (view) change.
- **Quiescence-ACK:** Market Server notifies the Group Leader that it is quiescent.
- **Quiescence-NACK:** Market Server notifies the Group Leader that it is not quiescent.
- **Ban:** Group Leader notifies Market Servers in its group that the specified agent (by id) is banned for one round.

References

[1] M. Babaioff and N. Nisan. Concurrent auctions across the supply chain. In *The Proceedings of the Third ACM Conference on Electronic Commerce*, pages 1–10, 2001.

[2] Shih-Fen Cheng, Evan Leung, Kevin M. Lochner, Kevin O'Malley, Daniel M. Reeves, L. Julian Schvartzman, and Michael P. Wellman. Walverine: a walrasian trading agent. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multi-Agent Systems*, 2003.

[3] Edsger W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[4] Se-Joon Hong. Supply chain management and electronic auctions.

[5] Friedemann Mattern. Global quiescence detection y based on credit distribution and recover.

[6] William E. Walsh, Michael P. Wellman, and Fredrik Ygge. Combinatorial auctions for supply chain formation. In *ACM Conference on Electronic Commerce*, pages 260–269, 2000.

[7] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[8] Michael P. Wellman, Daniel M. Reeves, Kevin M. Lochner, and Yevgeniy Vorobeychik. Price prediction in a trading agent competition. Technical report, University of Michigan, 2002.

[9] Michael P. Wellman and William E. Walsh. Distributed quiescence detection in multiagent negotiation. In *Proceedings of the Fourth International Conference on Multiagent Systems*, 2000.

[10] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: theory and implementation. *Decision Support Systems*, 24:17–27, 1998.

[11] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The michigan internet auctionbot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents (Agents-98)*, pages 301–308, 1998.