Peer to Peer Transactions in Agent-mediated Electronic Commerce

James E. Youll

Submitted to the Program in Media Arts and Sciences, School of Architecture and Planning, In Partial Fulfillment of the Requirements for the Degree of Master of Science in Media Arts and Sciences at the Massachusetts Institute of Technology

August 10, 2001

Abstract

This thesis proposes a new approach to electronic markets that overcomes the shortcomings of existing electronic markets through software agent-driven, peer-to-peer, iterative negotiations. Contemporary electronic markets commonly capture both the customs and shortcomings of traditional practice. Rule-based and bound to traditional models, contemporary electronic markets are overly controlling, segregated, inflexible, weakly automated and fixated on price. Many prior attempts to interpose electronic exchanges in existing markets have failed or encountered resistance from traders. Traders' resistance is interpreted here as a call for systems that bend to the will of traders while channeling self-interested actions toward healthy market practices.

The Atomic Market is both a model for an agent-based, peer-to-peer marketplace, and a working system that shows the model in operation. The Atomic Market architecture defines a decentralized marketplace wholly controlled by traders through a new protocol for distributed negotiation. The demonstration system is a collection of independent software agents that pursue economic exchanges via the Atomic Market methodology. In the Atomic Market, multiple agents resolve their self-interests though cycles of rewriting a "contract" comprised of descriptive, flexible terms tailored to the needs of each trader. The Atomic Market interprets the Contract Net Protocol as a message-passing system for economic negotiations, in which agents conduct broad, parallel searches to discover opportunities and trading partners in an open marketplace.

One of the first implementations of a decentralized, peer-to-peer agent marketplace, the Atomic Market brings together three features not found in contemporary e-commerce systems: decentralization, component-based transactions and open-ended outcomes. Buyers and sellers benefit from the exchange of detailed needs, offers, contingencies and external conditions as they define and negotiate both the substance and terms of a transaction.

Thesis Supervisor
Dr. Pattie Maes
Associate Professor of
Media Arts and Sciences
MIT Media Laboratory

Thesis Readers
Dr. Chrysanthos Dellarocas
Douglas Drane Career Development
Assistant Professor of Management
MIT Sloan School of Management

Dr. Benjamin Grosof
Assistant Professor of Information Technology
MIT Sloan School of Management

This research was funded by the Digital Life Consortium and E-Markets Special Interest Group at the MIT Media Laboratory, British Telecom, and MasterCard Corporation.

Notes

This edition of *Peer to Peer Transactions in Agent-mediated Electronic Commerce* was modified for redistribution. It differs slightly from the document submitted to the Program in Media Arts and Sciences during August, 2001, which will be archived at MIT as the official submitted thesis.

Specifically, this document has been altered for printing on single-sided laser printers, and for screen viewing. Blank pages have been removed, and the front matter and signature pages condensed. The table of contents has been re-rendered with single-spaced text. Bibliographic citations have been added to the preface. Finally, a copyright statement now appears in all page footers. Because of these changes, page numbers differ from those of the original thesis.

This document is available online at:

http://www.media.mit.edu/~jim/thesis/youll-thesis-2001-dist.pdf

BibTeX citation

```
@MastersThesis{youll-masters-2001,
 author =    {James E. Youll},
 title =     {Peer to Peer Transactions in Agent-mediated
             Electronic Commerce},
 school =    {Massachusetts Institute of Technology, MIT - Media Lab},
 address =   {Cambridge, MA},
 year =      {2001},
 month =     {September},
 URL =
{http://www.media.mit.edu/~jim/thesis/youll-thesis-2001-dist.pdf},
 comment = {S.M. Thesis.  Advisor: Pattie Maes},
 abstract =
      {This thesis proposes a new approach to electronic markets that
      overcomes the shortcomings of existing electronic markets through
      software agent-driven, peer-to-peer, iterative negotiations.
      Contemporary electronic markets commonly capture both the customs
      and shortcomings of traditional practice. Rule-based and bound to
      traditional models, contemporary electronic markets are overly
      controlling, segregated, inflexible, weakly automated and fixated on
      price. Many prior attempts to interpose electronic exchanges in
      existing markets have failed or encountered resistance from traders.
      Traders  resistance is interpreted here as a call for systems that
      bend to the will of traders while channeling self-interested actions
      toward healthy market practices.

      The Atomic Market is both a model for an agent-based, peer-to-peer
      marketplace, and a working system that shows the model in operation.
      The Atomic Market architecture defines a decentralized marketplace
      wholly controlled by traders through a new protocol for distributed
      negotiation. The demonstration system is a collection of independent
      software agents that pursue economic exchanges via the Atomic Market
      methodology. In the Atomic Market, multiple agents resolve their
      self-interests though cycles of rewriting a  contract  comprised of
      descriptive, flexible terms tailored to the needs of each trader.
      The Atomic Market interprets the Contract Net Protocol as a message-
      passing system for economic negotiations, in which agents conduct
      broad, parallel searches to discover opportunities and trading
      partners in an open marketplace.

      One of the first implementations of a decentralized, peer-to-peer
      agent marketplace, the Atomic Market brings together three features
      not found in contemporary e-commerce systems: decentralization,
      component-based transactions and open-ended outcomes. Buyers and
      sellers benefit from the exchange of detailed needs, offers,
      contingencies and external conditions as they define and negotiate
      both the substance and terms of a transaction.}
}
```

Contents

Acknowledgements

You've got to jump off the cliff all the time
and build your wings on the way down.

– Ray Bradbury

My gratitude is humbly offered to those named below, and to all who are not, who have shared and sometimes carried me through exciting, difficult, happy and sad times. The archivists say an image of this page will be burned onto a little sheet of microfilm that someone might see, someday, maybe. Other than a shot at eternity in the MIT thesis archives, I just don't know how else to thank everyone.

There is no way to convey the magnitude of my debt to Professor Pattie Maes, who gave me the opportunity to fulfill a lifelong dream. I still don't really believe I have an MIT ID card and Media Lab keys in my pocket. Yet here I sit writing the last few words on the last page of two years of work with and for Pattie, as her last Software Agents graduate. Thank you for advice about technology and life, for your confidence in us all, for your patience through innumerable edits of this difficult document, for your trust, for leading by example, for reassuring words, and for knowing better.

Heartfelt thanks to Professor Hiroshi Ishii for advice, counsel, encouragement and support. I am honored to have been your student, and bettered by all you have taught me.

Thanks to thesis readers, Professors Benjamin Grosof and Chrysanthos Dellarocas, for caring enough to make this work challenging and rewarding.

Thanks forever and ever to Professor Ann Marie Lancaster, who probably doesn't even know how much her support has meant to me over many years and many careers.

Thank you, Joe, for daring me, for believing this could really happen, for saying "I told you so" when it worked out, and for supporting a personal adventure turned into a shared adventure, with too many deliberate and accidental disasters.

Thank you, best friends and steadfast supporters
Steve and Debbie Youll; Dave, Krista and Brian Youll; Mary Leinhos, Dave and Julie Kuhar

Thank you, to so many who made this possible

Hal and Barbara McLean, who think anything is possible

Joe and Sally Wright, who think nothing is impossible

Don and Agnes Gray, for moral support and perspective

Kent Libbe, a model of calm and confidence

Professor Ronald Lancaster who waited for me in a snowstorm when he didn't have to

Karen Thompson who shepherded me through personal storms when she had better things to do

Thomas Karovic who gave me his confidence and trust when I was too young to be handed either, opening doors I would never have found on my own

Professor Walter Maner who knows computer science and philosophy really are the same thing

Professor James Gordon for pushing, encouraging, and demanding excellence

Nicholas Negroponte, Jerome Wiesner, Muriel Cooper, Walter Bender, Linda Peterson, Deb Cohen, Kate Shanaghan and all the good people in CASR, and all the pioneers who built the Media Lab, who make it work now, and who found a small place in it for me.

Sr. Joanne Caniglia who drove me two hours to see "a computer," back when there were no "personal computers" and as far as anyone could tell, no need for them. I memorized the green-bar printout of the little BASIC program we wrote that day.

My mother and grandmothers for their love and for everything that grandmothers and mothers do… my grandfathers who made it possible for me to start my first business, by having had the courage to start theirs. I wish they were here to see that things worked out okay. And thank you so much to my "adopted" parents, Al and Nancy Meiring, two of the most good-hearted people I have ever met.

Thank you old friends

Dan Spitler, Linda and Buzz Liber, Bev Nathan, Susan Pauly, Jim Nitchals, Peter Meyers and Charles Terenzio. Thanks to the writers, editors and shooters including Joe Giampietro, Luwayne Tompkins, Ray Lee, Steve Ilko, Rob Engelhardt, Mike Semple, Bill and Joyce Lewis, Robert Miller and Sal Marino who helped me become a real photojournalist before I was old enough to be a real photojournalist.

Thank you new friends, collaborators, teachers and unindicted co-conspirators

Joan Morris – I guess things worked out okay even after that strange first day here. Thanks for putting up with my eccentricities, not to mention the junk in our shared office.

Agents – Sybil, Henry, Raffi, Nelson, Brad, Mike, Emily, Gaurav, Tucker, Dennis, and Mukul

MITians – Joe Paradiso, Adam, Mike, Wendy, Jofish, Dana, Natalia, Vanu, Kristin, Push, and Professors Judith Donath, John Maeda, Glorianna Davenport, and Ted Selker

UROPs who toughed it out with me over the last two years: Ralph Harik, Pauline Hsu, Jon Lee, Lee Lin, Sanjiv Parekh, and Aleksandra Szelag. Thanks for great collaborations and a lot of exhausting fun. Brilliant futures to you all!

Good people whose hearts and deeds redeem our humanity

Dr. Irene Pepperberg, Dr. Paul Constantino, Susan Farlow, Dr. Marge McMillan, Pam Sherwood

and inspirations

Bono, Edge, Larry and Adam; Waters, Mason, Gilmour, Wright and Barrett; Gabriel; Berry, Buck, Mills and Stipe; Miles; Stanley; Seuss

This work is dedicated to Nancy Meiring
who saw the beginning of this adventure
but not the end.
We miss her terribly.

Press any key to continue.

1 Introduction

> *In the long run, even the most brilliant and unprincipled huckster must expect ruin; but let the hucksters become tired of this and agree to live in peace with one another, and the great rewards are reserved for the one who watches for an opportune time to break his agreement and betray his companions. There is no homeostasis whatever. We are involved in the business cycles of boom and failure, in the successions of dictatorship and revolution, in the wars which everyone loses, which are so real a feature of modern times*.
>
> *– Norbert Wiener* [1]

This thesis introduces the "Atomic Market," the name given to both an architectural model for an agent-based, peer-to-peer market system, and to a working demonstration system that shows the model in action. The Atomic Market *model* describes a marketplace for software agents that is (1) decentralized, (2) component-based and (3) open-ended. The Atomic Market *demonstration system* is a collection of independent software agents that receive, modify and retransmit messages in pursuit of economic exchanges via the Atomic Market methodology.

Atomic Market agents extend the Contract Net Protocol (CNP) [1]. The CNP is a decentralized approach to coordination in a multi-agent system. It is a straightforward method by which software agents recursively seek out peers, delegate tasks, and generate a network of cooperating agents with both top-down control and decentralized delegation. Previously proposed extensions to the CNP [2] have included *bidder feedback*, *mutual monitoring* to dissuade bad behavior, and *decommitment options* to permit a trader to back off a committed task if presented with a significantly better alternative.

Atomic Market agents use parallel, iterative conversations rather than one-by-one solicitations to carry out broad searches for favorable terms and trading partners. The Atomic Market relaxes the CNP's linkage between *manager* and *contractor*. In contrast to the CNP, Atomic Market agents do not assign *tasks* to one another, but rather exchange component-based *draft transactions* in the spirit of the CNP's recursive distribution scheme. Through peer-to-peer exchanges, agents discover market prices, search, negotiate, and carry out all the activities of "market participants" by altering the drafts

---

[1] *Cybernetics; or, Control and communication in the animal and the machine*, The M.I.T. Press and John Wiley and Sons, Inc., 1961.

of proposed final transactions, called *market messages,* as they exchange them. A market message encapsulates a draft transaction, common terms of the deal (such as an overall deadline binding on every trader), and a small amount of state data.

Decentralization

The traditional central control of a *market authority* is absent here. A fundamental presumption of the Atomic Market is that self-interested software agents, interacting on behalf of people and companies, can cooperatively draft economic transactions without the assistance, interference or oversight of a central market authority. Even more, there is no central *place* or *server* in the system. Agents communicate directly in a peer-to-peer fashion. The only central *place* in the Atomic Market as currently designed is a weak directory called the *registry* that provides a yellow pages-like service. Ultimately, this lone central component could also be replaced by a distributed directory. The rationale behind the *registry* is discussed in more detail in Chapters 4 and 7.

Component-based transactions

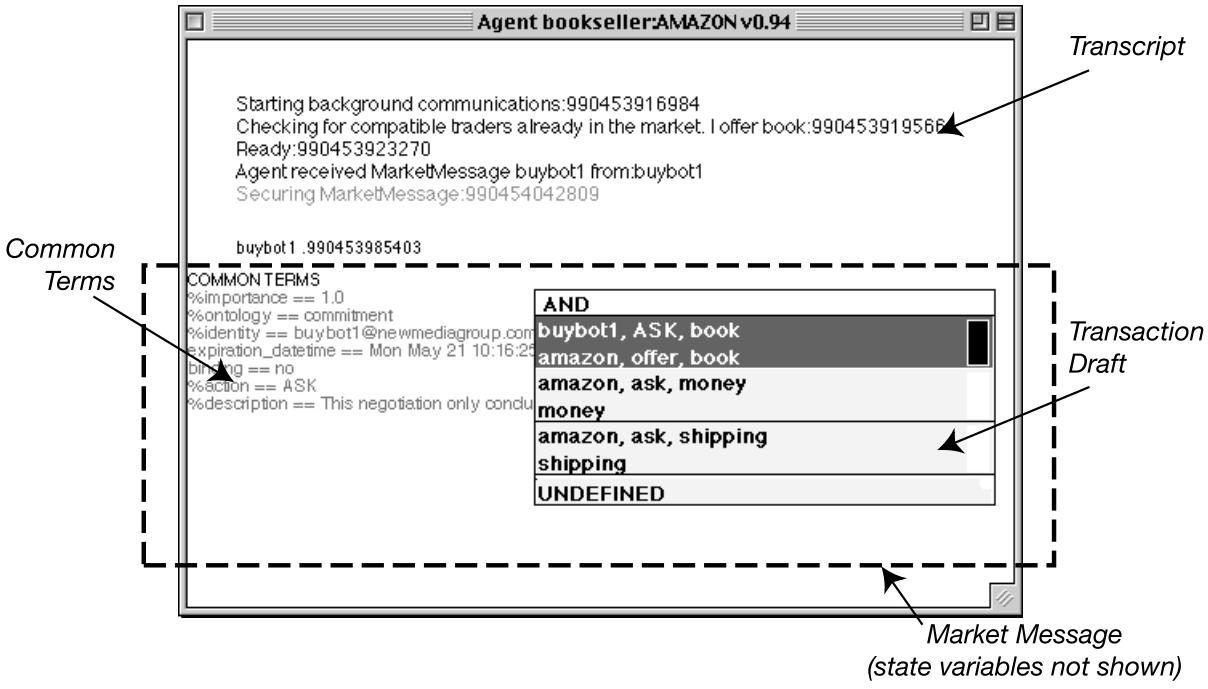


Figure 1-1: Book-selling agent monitor, showing transaction draft, common terms, and activity transcript. The common terms and transaction draft together with state data (not shown) comprise a *market message.*

The component-based or *composite* transactions [3] exchanged by the agents are collections of individual offers and needs representing larger real-world transactions. For example, even a mundane

activity such as a *book purchase* requires a *book*, *shipping*, *payment* and *insurance* – possibly from four distinct providers. Each of these items is a component of the greater transaction *book purchase*. In the Atomic Market, components of a large transaction, e.g. *book* and *shipping*, can be separately negotiable. Further, non-traditional, highly descriptive components such as *tomorrow's weather forecast for Boston*, *the current retail price of this item*, or *the number of people who order this item by 5pm today* can all be integral components of Atomic Market transactions. In general, Atomic Market *components* are smaller and more precise than the ordinary "smallest negotiable parts" of conventional transactions. Because components are so small and may be individually bid on an open market, new investment options could be spawned. For example, an investor could conceivably buy up the insurance coverage for *all FedEx shipping scheduled between 1pm and 2pm*, profiting when packages arrive safely, and paying out if they do not.

Based on private knowledge, an agent may try to satisfy the needs expressed by other traders' components of a market message, or the agent may add its own components, inviting still more traders to join the deal. Thus, the "task assignment" of CNP is borne in the implicit duty of an agent to alter a market message if it is able. The agent may complete the transaction draft by answering the unmet needs of others. It could also expand the message, adding missing components to nudge the message toward completion. For example, if a bookseller knows that shipping and insurance are needed to complete an order, it can add shipping and insurance components, thereby inviting shippers and insurers to tender bids. This iterative *transaction design* phase encapsulates and extends the *product brokering, merchant brokering* and *negotiation* phases of the buying behavior model previously defined by Maes, Guttman and Moukas in [4].

Open-ended outcomes

The marketplace is *open-ended* because little about the final fate of a market message may be predicted a priori. A market message changes dynamically as it wends from agent to agent, either drifting toward a final state or disappearing altogether if discarded by a disinterested trader. Any agent may edit a market message by adding, removing or changing the components of its transaction draft and common terms. These alterations are driven by individual agents' private strategies, which may shift dynamically in response to the current content of the draft, market activity, or external conditions known and sensed by an agent. Neither the success, nor failure, nor final content, nor ultimate roster of participants on a market message is certain until the last component has been filled and the message "completed."

Motivation

The three characteristics that distinguish the Atomic Market, *decentralization*, *component-based transactions* and *open-ended outcomes*, are not features of most contemporary e-commerce systems. Rather, existing electronic markets have incorporated both the customs and the *shortcomings* of prior practice. They are overly controlling, segregated, inflexible, weakly automated and unnecessarily fixated on price. Electronic markets' adherence to convention is understandable. Established traders have already rejected inflexible, technology-based alterations of familiar practices even when those changes might have aided fractured, inefficient markets [5]. Traders have also maligned upstart exchanges as "unwelcome outsiders" that divert control and profit from incumbents [6]. So great was the enmity of European paper traders, for example, that three nascent paper exchanges were driven to close or cede control to the industry.

Clearly, a self-interested market cannot interpose itself in an established flow of trade without an invitation. However, resistance to externally imposed changes in *practice* need not imply that existing practice is the only viable model for electronic commerce. Rather, this thesis interprets the resistance as a call for new systems that bend to the will of traders while offering novel modes of interaction that are optimized for electronic commerce. Such systems should be compatible with both the self-interests of traders and with healthy market practices. The Atomic Market proposes an approach that answers this challenge via a marketplace wholly controlled by traders.

Contributions

This thesis interprets the CNP as a message-passing system for economic negotiations. Taking advantage of the high bandwidth and processor speeds available today, agents undertake parallel rather than serial negotiations. They are less tightly bound than the agents in a traditional contract net, and pursue private goals through iterative, parallel rounds of contract proposal and modification. The contributions of this thesis include

1. One of the first implementations of a decentralized peer-to-peer agent marketplace.
2. A new protocol for distributed peer negotiation.
3. A theory and system for resolving the self-interests of multiple agents through iterative cycles of rewriting of a "contract."
4. Implemented examples that demonstrate the advantages of such a system.
5. A critical discussion of the implications and limitations of peer-to-peer solutions in comparison to centralized solutions.

Overview

Chapter 2 discusses limitations of present-day electronic markets. A theory of convergence toward component-driven, decentralized, peer trading environments is built on this discussion, with a plan by which a more flexible market system may emerge from peer-to-peer agent interactions.

Chapter 3 reviews related work, the companies and standards whose efforts shape present-day Internet commerce, and the foundations of the Atomic Market: multi-agent systems, economics, self-organization, social welfare theory and negotiation.

Chapter 4 introduces the Atomic Market, a working model of the theories of peer-to-peer commerce of Chapter 2. The chapter provides an overview of the Atomic Market and its traders, through comparison to traditional markets, a detailed example of peer messaging between two agents, and a brief review of the architecture of the agents.

Chapter 5 considers risk factors for traders in a peer-to-peer marketplace, and the design decisions these risks drove in the creation of the Atomic Market.

Chapter 6 discusses a theory of markets-as-networks, an approach that was used to simplify the design of agents and to shore up protections against the risk factors discussed in Chapter 5.

Chapter 7 details the implementation of the Atomic Market, including sample code and review of the capabilities of a generic agent trader. The chapter presents considerable detail about the structures of Atomic Market elements, and operations that may be performed on them. Readers with a strong interest in implementation details may wish to use Chapter 7 as a standalone reference while reading other parts of the thesis.

Chapter 8 concludes the thesis with a discussion of the issues of software agents in electronic markets. It considers the forces that stand in opposition in an Atomic Market, resource consumption and capacity issues of the demonstration system, the practicality of commercial deployment of such a system, and finally future avenues of exploration and improvement.

2 Electronic Markets

Constraints, complications and opportunities in electronic markets

When developers of contemporary e-commerce systems mapped manual practice to digital form, the customs, rules and even shortcomings of that practice were mapped as well. As the most egregious of the ugly commerce models such as "online malls"[2] have begun to evolve, consolidate or disappear, e-commerce practitioners can rethink trading systems. This section will discuss some problems with present-day electronic markets, and will raise questions about those shortcomings that motivated the Atomic Market project and this thesis.

Too much control

As with a conventional market, a traditional electronic market enforces conventions that create a stable, predictable environment for trading. Markets take these measures because they acquire and maintain relevance only by the consent of traders, consent granted as a measure of each trader's confidence in the market. Yet a trader's true interest is not the market itself, but the offers and needs of other traders. Despite its role as matchmaker and deal facilitator, a traditional market may even interfere with trading. For example, a market sets the rules by which traders may interact, the hours during which they may trade, even – in the case of equities markets and some online auctions – the price increments at which they may buy or sell. A traditional market defines the lowest common mode of interaction even for trades that traders see as unique, discrete, private events. Powerful markets aggressively punish those companies whose presence does not benefit the market. For example, NASDAQ *delists* companies that don't meet share price or capitalization requirements. Delisting can "put companies into a death spiral" [7].

In summary, a traditional market decides on the pairing of buyers and sellers, the negotiation mechanisms they may use, and the language used to describe products and features. All these decisions, made for the market's sake or even of operational necessity, constrain traders.

---

[2] e.g. The Free World Mall, http://www.freeworldmall.com/index2.htm

Segregation

Traders in a particular electronic market cannot see trading partners in other similar markets without first joining those markets. Their access to comprehensive product and price information is thus limited. For example, Chemdex, PlasticsNet, e-Chemicals [8] and ChemConnect [9] are all online marketplaces for buyers and sellers of chemicals, gases and similar industrial goods. However, traders must join each marketplace separately and manually toggle between them to glimpse the entire *market* for a particular item.

Inflexibility

Competing for-profit electronic markets distinguish themselves through proprietary trading environments. For example, Priceline.com invites customers to "Name Your Own Price.[SM]" Before they failed, Mercata.com and Mobshop.com aggregated demand by lining up many customers in advance then selling the goods in bulk at a discount [10].

However, buyers cannot consolidate several individual orders for airline tickets at Mercata.com (nor *rent* the Mercata model), then bid for the tickets at Priceline.com. Traders also cannot ordinarily express contingent terms for electronic trades, even when the terms would be readily understood by a human counterpart. For example, a trader may wish to order "a picnic lunch for 10 people, if the chance of rain tomorrow is less than 30 percent," or "rock salt, if you can supply at least 60% of the ordered amount." Some individual trading systems might support subsets of this functionality, but no system allows a trader to place a contingent bid with, for example, the *entire* rock salt market in one move. Nor can a trader attach arbitrary contingencies not anticipated by the system's designers.

Monolithic interaction models

E-commerce systems and standards presume that electronic transactions can be fully formalized in standards. For example, the Internet Open Trading Protocol [11] defines transactions between one "merchant" and one "consumer." Generic XML approaches are also not prescriptive of a solution. An XML Document Type Definition (DTD) rigidly represents its designer's view of the proper form of a transaction. What is really needed is a process to permit transactions of all kinds to evolve from component parts assembled by the traders themselves.

Weak automation

Electronic markets that merely transfer manual practice to online form cannot encode nontrivial requirements. For example, FreeMarkets [12] and CommerceOne [13] manage human-controlled trading systems. Decisions in these systems are made by people, not software agents. Similarly, the automated bidder's proxy at eBay can increase a bid without human approval, but cannot compare the current bid to retail prices outside eBay to protect a buyer from paying too much.

Emphasis on price

Despite empirical findings that customer awareness, branding and trust, not price, drive Internet purchase decisions [14], online auction mechanisms generally use *price* as the key point of negotiation. Automated price-shopping services abound, but few attempt the difficult task of integrating a customer's true desire with buyers' offers.

Disintermediation may not be such a great plan after all

One hope of Internet e-commerce was that direct merchants would profit by the elimination of middlemen – those traders who are simply go-betweens in the "food chain" from producer to consumer. This potential profit source was perhaps over-estimated. While disintermediation does eliminate the dispersal of profits it also transfers shared costs and shared risks formerly absorbed by intermediaries back to principals. Disintermediation also eliminates the intermediaries' reputation- and market-building services [15]. Internet-based, intermediary-free principals now find themselves responsible for managing entire markets without help on the front lines, alone in competition against established firms represented by armies of middlemen. Since the beginning of the Atomic Market project, many companies that attempted to frame novel *market models* in proprietary, intermediary-free, closed *market places* have failed. Among the casualties were demand aggregators Mercata.com and Mobshop.com.

Moving beyond order execution

Electronic marketplaces as currently deployed are fundamentally *order execution system*s rather than proactive tools to connect buyers and sellers. The electronic commerce systems that run these markets tend to model the paper flow of transactions. So doing, they may thwart apparently reasonable requests. For example, a colleague recently tried to buy gifts for two friends online. The sale was lost because the system forbade sending gifts to more than one address per day, per customer. Rather than treating every interaction and customer individually, it applied a global worst-case rule even to a

long-term, regular customer. This transaction would have gone through without a hitch in a catalog store twenty years ago, but was impossible on the Internet last Christmas.

Real traders frequently issue bids and requests that would not fit into conventional electronic market systems. We might *book a flight if it leaves Saturday morning, costs less than $200 and has a window seat available.* We might *buy baseball tickets if the weather forecast is 'sunny and warm' and at least three friends respond to the invitation,* or *swap some paperbacks for a DVD* without using money. To pursue "complicated" trades like these today, traders must work partly inside the electronic market, and partly outside it. For example, the baseball ticket buyer would await replies from friends through Evite[3], then check a weather forecast, and finally buy the tickets if all conditions are met.

Electronic markets' inflexibility and adherence to traditional practices was perhaps unavoidable in early systems. However, though traders have rejected inflexible, technology-based alterations of their preferred familiar practices, their resistance highlights an opening for systems that are compatible with both the self-interests of traders and with healthy market practices.

[3] http://www.evite.com/

The Atomic Market

Could automated market technologies ever permit reasonable traders with reasonable expectations to trade on precisely their own terms, entirely within a market, even if the arrangement might seem "unconventional?" The Atomic Market is an attempt to answer that question with a market defined by the policies and actions of its traders rather than through the decrees of market management.

The Atomic Market leverages three features to facilitate "complicated" transactions such as those described above to offer traders more flexible and robust interactions.

First, the interactions between traders are *peer to peer*, not routed through a central authority. Traders are free to construct any plausible transaction. However, in gaining this freedom, they also lose the protections afforded by centrally managed markets. The Atomic Market proposes and demonstrates some possible means of providing these assurances.

Second, transactions are *component based*. That is, all the separable features of a transaction can be negotiated individually. The weather forecast, invitation responses and tickets in the baseball example previously mentioned would probably come from three unrelated providers, perhaps with separately negotiable fees. In the Atomic Market, diverse sources may be commingled without advance coordination. For a second example, consider a shipper quoting a real time price for the transport of a single book from Amazon.com to a customer. The shipper would consider the capacity remaining on pick-up trucks at the source, the density of parcels expected to move through its hub, and the capacity left on delivery trucks at the destination before assigning a cost to the delivery.

Finally, negotiations are *open-ended*. We cannot predict how (or whether) a negotiation will terminate because the private strategies and actions of the traders may change as the negotiation progresses. Traders may add terms to a transaction, and can even invite others to join a negotiation.

In contrast to traditional expectations for electronic commerce, Atomic Market interactions may be heavily intermediated. Agents may deal with one another through many (not always evident) layers of indirection. Strictly speaking, the Atomic Market may disintermediate traditional middlemen such as traditional retail sellers. However, it also facilitates the new kinds of intra-market traders such as for-profit information and recommendation services.

Overview

The main functional parts of the Atomic Market system are:

Registry – A directory and event service supporting the discovery of trading partner

Trading agent – Software that processes transactions in accordance with built-in goals, strategies and needs

Market message – A collection of atoms, tied together with propositional logic, that describes the features of an under-construction transaction in fine detail

Atom – The basic component that describes one feature of a market message

The registry

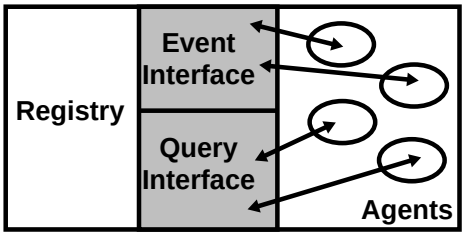


Figure 2-1: Agents and registry Interface

In the Atomic Market, the role of a central "marketplace" is significantly diminished compared to other markets. The only remnant of a traditional, centralized market is the registry, an event-driven directory service for discovery of potential trading partners. Events ("notify me when a compatible trader arrives") and queries ("are compatible traders present?") are supported. The registry simply notifies, for example, *sellers of books* that a possibly compatible *buyer of books* has asked to meet new trading partners. The registry does not offer recommendations or filtering. This approach is compatible with the proposed Universal Description, Discovery, and Integration (UDDI) [4] specification. Ultimately, it could be replaced by peer-to-peer directories for a completely decentralized system.

[4] http://www.uddi.org/

Trading agents

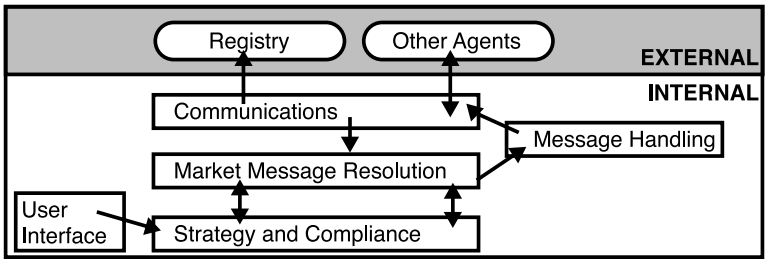


Figure 2-2: Schematic of a Trading Agent

A trading agent (Fig. 2-2) processes market messages with an overall goal of converting them into completed transactions. The *market message resolution* subsystem within an agent works with the agent's *strategy and compliance* component to apply the agent's goals, strategies and needs to ongoing negotiations.

Agent-to-agent messages

In the multi-agent Atomic Market system, an agent in search of a good or service composes a draft of a desired transaction, called a *market message*. The market message describes the things the agent wants to buy or sell. It could contain contingent terms or call for third-party data from diverse sources such as weather reporting agents or credit bureaus.

As illustrated in Figures 2-3 through 2-7, the agent sends this *market message* to traders that it believes can answer its needs. If the agent wants to meet new trading partners, it also posts a solicitation (e.g. "I want a book") to the registry. Once they have met, traders communicate directly with one another. They handle the market message in a back-and-forth collaboration as each tries to *complete* the transaction. For example, when a book seller agent receives a message that asks for a book, it may answer the "need book" message by adding "offer book AND need money AND need shipping. As the message is passed around, other traders (e.g. shippers) become involved. The message becomes more complex but also more descriptive of the needs of all the traders. Eventually, if every need is met, the message stabilizes, the negotiation ends and the transaction can be executed.

The basics of inter-agent communications – message structures, and to a lesser extent the protocols for exchange – must be formalized and adopted by all trading agents. All other market activities, including the terms of commitment, are entirely the province of the individual traders. The Atomic Market system demonstrates the use of formal message structures and protocols in an otherwise unregulated peer-to-peer setting.

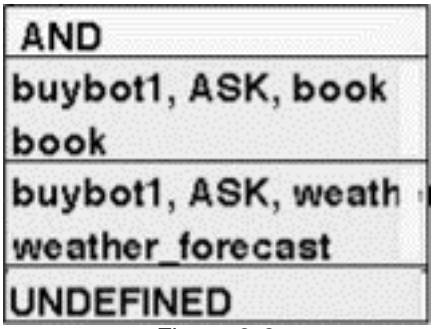


Figure 2-3

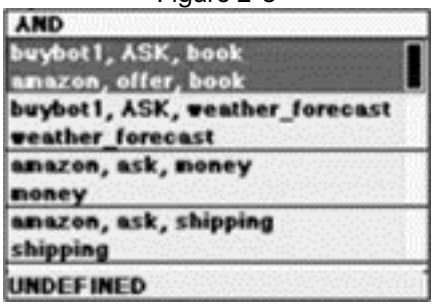


Figure 2-4

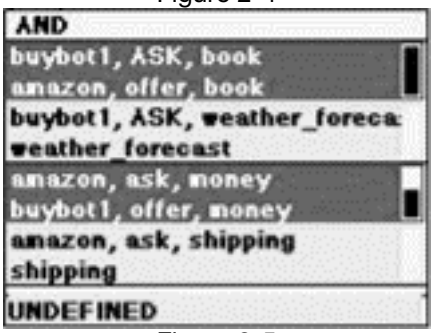


Figure 2-5

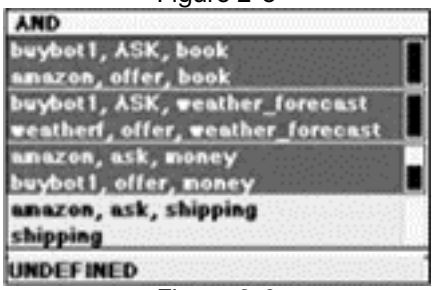


Figure 2-6

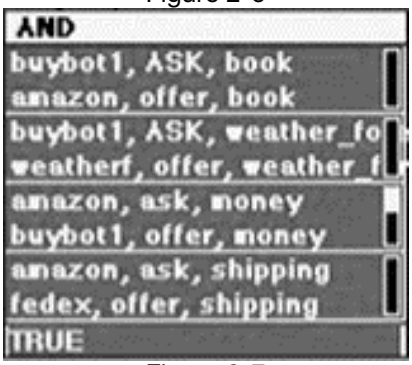


Figure 2-7

Agent-to-agent market message handling

1. The book-buying agent *buybot1* drafts a market message. For testing, we made the order contingent on a sunny weather forecast. The agent advertises in the registry for agents that can fill book orders. It meets the *amazon* agent, and sends it a copy of the market message.

2. The *amazon* agent fills the request for a book, and adds two terms of its own: a request for money, and a request for shipping services. It then sends the message back to *buybot1*.

3. *Buybot1* fills *amazon's* request for money, and then must decide where to send the message next. Here, it forwards the request to an agent that can handle the first unfilled request – a weather forecast. The book-buying agent either knew the weather forecast agent in advance, or advertised a need for a weather forecast in the registry.

4. The weather forecast agent *weatherf* supplies a current forecast then must decide where to route the message. It has no strategies for locating shippers (and no interest in doing so) so it hands the message to the agent that created the shipping requirement – the *amazon* agent.

5. The *amazon* agent solicits shippers through the registry. Upon meeting the shipping agent *fedex* it sends that agent a copy of the market message. *Fedex* fills the request for shipping services, completing the market message. Once the last component has been filled satisfactorily, the message status becomes *true*. Fedex broadcasts the market message to all participating traders.

Presumptions of the Atomic Market

1. Agents are competitive. However, they must cooperate to a limited extent or no trades can succeed.

2. An agent may add components to a transaction, increasing its complexity and specificity. An overly specific transaction may be difficult to complete. An agent may also remove terms from a transaction. However, a too-simple transaction may not recover the maximum utility for the trader because it fails to convey the trader's true, detailed preferences.

3. When an agent receives a market message, it will attempt to resolve all unsatisfied components, especially those it created. For example, a book-buying agent should provide payment if asked to do so by a bookseller. Every agent implements private resolution strategies. Only the agent knows its strategies, and those strategies may change as a negotiation proceeds.

4. The distributed search spawned by traders leads to high redundancy. Many possible paths will be explored. Market messages circulate until they are lost, discarded or completed. A lost message reduces the number of possible paths from request to resolution.

5. Agents are not required to understand or approve transaction terms that do not affect them directly.

6. All agents calculate cryptographic signatures over messages through a globally uniform algorithm. These signatures protect the message "in plain sight." An agent may discard any market message in which its own components have been altered by others, or it may correct unsatisfactory alterations and propagate the message as usual.

7. Any agent may alter a circulating market message. In fact, there is no protection against such alterations in a decentralized system. The cryptographic signatures reveal both authorized and unauthorized changes to messages. Issues of security, trust, message protection and specific risks are discussed in detail in Chapters 4, 5 and 7.

8. For ease of implementation in the Atomic Market demonstration system, agents share a common set of basic hard-coded ontologies that represent the available components of transactions.

3 Related Work

A thumbnail sketch, a jeweler's stone
A mean idea to call my own
Old man don't lay so still you're not yet young
There's time to teach, point to point,
Point observation, children carry reservations
Standing on the shoulders of giants leaves me cold, leaves me cold.
A mean idea to call my own, a hundred million birds fly

– R.E.M. [5]

The Atomic Market draws inspiration and guidance from several subject areas, including distributed artificial intelligence, agent coordination theory, welfare economics and contemporary electronic commerce. Resonant among these are issues in the design of interaction protocols for *machines that make deals*, as discussed in Rosenschein's *Rules of Encounter* [16] which proposed the creation of social environments that coax optimal, beneficial behavior from machines. Rosenschein called this *social engineering,* or "helping designers establish an automated society's rules."

Rosenschein also identified several related areas in AI research including *planning, axiomatic solutions* and *social laws.* These will be discussed next, followed by a review of *previous experimental agent systems*, *electronic commerce practice*, *agents in e-commerce,* and related *standards and initiatives*.

[5] R.E.M., *King of Birds*, Document, released Aug. 31, 1987, IRS Records, IRSD-42059

Distributed Planning

Planning concerns the methods by which agents may avoid conflict and advance together toward a mutually satisfactory solution to a partially shared problem. Planning requires advance effort and perhaps ongoing compromise, plus coordination and communication as the agents work toward a solution. *Distributed planning* is an approach through which multiple agents work together to reach decisions about their collective future actions.

Some agents may be naturally *collaborative*, even sharing goals and an interest in mutual success. In some domains, these agents readily yield to one another when compromise helps bring about a desired outcome for the group. The team-based entrants to RoboCup robotic soccer tournaments [17] exemplify distributed planning by collaborative agents. Furthering these goals, the metaphor of team-oriented programming [18] defines an API to facilitate the design of agents that participate in team-based activities and joint planning.

The Atomic Market presumes that agents are competitive but cooperative. They may be interdependent in the pursuit of complex personal goals, but they cannot be expected to act altruistically "for the good of society."

Contract Net

Much of the fundamental work in planning for multi-agent systems builds on Smith's Contract Net Protocol (CNP) [1]. The CNP defines a decentralized method by which self-interested ("SI") agents jointly complete a complex, decomposable task through a process of bidding and work assignment, called *task sharing*. The protocol is a high-level definition unconcerned with physical architectures and implementation details. It defines the flow and character of inter-agent coordination and work distribution. A *Contract Net* is a coalition of agents that work together to complete a task using the CNP.

Briefly, contract net participants decompose large tasks into parallel subtasks, then *advertise* the subtasks to find other agents that are willing to complete the subtasks. Interested, available contractors answer advertisements with binding *bids.* The *manager* that placed the advertisement then *awards* each *contract* to a selected *contractor*. The contractor is expected to complete the assigned task. When an agent accepts an assignment, it may divide its task into further subtasks, then advertise

those subtasks in the same manner that the original task was advertised. Thus, a contract net agent may act simultaneously as both manager and contractor.

Contract net agents, called Knowledge Sources (KS) in the original text, are *loosely coupled.* They work asynchronously toward the completion of subtasks spawned from the subdivided main task. A manager may interact with a contractor while it is working on a task, to receive progress reports, to cancel an in-progress task, or to receive results of the contractor's work.

Architectures that follow a contract net model, such as the Atomic Market, cannot presume that participating agents are cooperative or working as a team, even though they may ultimately depend upon one another to achieve their goals.

Problems with the contract net

The original Contract Net Protocol defined the "connection problem" as a significant issue in decentralized planning. The problem has two components: *resource allocation* and *focus*.

Resource allocation is the problem of balancing computational load throughout delegates by distributing parallel tasks effectively to maximize throughput.

Focus concerns the decomposition of tasks into subtasks and the selection of strategically beneficial Knowledge Sources. Because the "most appropriate" KS for a given subtask cannot be known a priori, an agent (or KS) should try to discover the most appropriate interaction partners. However, the agent cannot approach every potential contractor with every need. The recursive nature of the contract net would trigger a combinatorial explosion of messages. In both the Atomic Market and the CNP, agents must contact potential trading partners selectively. If agents are not selective in their communications with others, the free message transport may lead to a *tragedy of the commons* [19] and degradation of service quality for all. Even the original CNP suggested that an agent that transmits a message should expect a message in return, and so should regulate its own messaging to limit incoming traffic. However, stronger, more provable methods have been proposed in extensions to the contract net, such as those discussed next.

Extensions and adaptations of the Contract Net protocol

A difficulty with contract nets of self-interested agents is that the agents may not always disclose truthful information, confounding optimal global task allocation. An agent's local decisions may also be sub-optimal. The agent cannot accurately compute the value of accepting a task without a view of all possible tasks – those available at the moment and those that might arrive in the near future – but that information is not available. A further commitment-related problem is that when the first of two parties makes a binding commitment, the second party knows the contract will definitely be made if it also commits. Thus the relationship is not symmetrical. One agent must bid not knowing the outcome. The other has full knowledge.

Sandholm proposed a number of extensions to the contract net protocol [2] including more fluid negotiations to allow the agents to adopt varied levels of commitment during the negotiation. Leveled commitment options can allow a more thorough search as agents are free to make contingent commitments, but may back away ("decommit") if a substantially better offer arrives. With decommitment terms in place, an agent may test one offer with many agents, allocate risk according to its own risk tolerance, and enter contingent contracts. Additionally, decommitment accommodates linked items in contracts. Early CNP items were negotiated one at a time – insufficient for many activities. The proposed extensions permit the readjustment of tasks upon receipt of new information.

Sandholm also discussed accept/terminate/counterproposal feedback messages to managers from prospective contractors. However, a contractor has little incentive to transmit a *termination* message unless it is "mostly negotiating" with a particular manager. A contractor is *discouraged* to reply when dealing with a competitor, if the reply would help the competitor, even though the free flow of feedback would improve overall efficiency.

As for commons problems, both economic and peer-management methods can be used to dissuade agents from overuse of shared resources like cheap message transmission. A technique proposed by Sandholm is *mutual monitoring*, so the agent society can flag violators. This might occur via an out of band control channel or through a distributed reputation mechanism such as [20] implemented as a market service.

An economic approach to the commons problem proposed by Sandholm and others to prevent agents from generating excess traffic is the levying of a small fee by message recipients upon message senders. Senders "pay" recipients to consider their offers. Pricing these payments can be a difficult

affair. A well-funded agent designed to dominate the market might "spam" other traders unimpeded by what is to it a minor expense, while beneficial agents might lack the currency to introduce themselves to others.

Parkes proposed a fully decentralized mechanism that efficiently moderates messaging in an automated society through payoffs between agents [21]. Dignum offered an approach to the CNP for open environments in which agents could use *persuasion* to form teams to collectively solve a problem [22]. In that flexible architecture, ordinarily independent agents work as team members until the collective goal has been fulfilled.

Bridging a gap between implementation-oriented cooperative problem solving (CPS) models and CPS theory, Wooldridge presented an abstract formal model for cooperative problem solving in a four-stage process consisting of *recognition*, *team formation*, *plan formation*, and *action*. [23]

The form of the agent-to-agent contract itself is addressed by declarative approaches to business rules in contracts, and a refined form known as Courteous Logic Programming (CLP). CLP is a means of representing business rules in e-commerce contracts [24] [25], including provisions for overrides and prioritization of conflicting rules.

Comparison of contract nets and the Atomic Market

Trading partners in an Atomic Market *transaction draft* are analogous to nodes in a contract net. A contract net's *tasks* are similar to the *components* of an Atomic Market transaction draft. Atomic Market *traders* delegate work to peers as the *managers* in a contract net subdivide labor to *contractors.* These similarities reveal how the Atomic Market follows the original contract net in principle. The following discussion will step through some features of the original contract net as well as previously proposed extensions, to discuss how the Atomic Market's approach to the activities within the society of collaborators differs from past practice.

In the original contract net, every node could divide tasks as necessary, and could solicit others recursively to assign tasks. Atomic Market traders similarly solicit assistance from others, but do not *assign* tasks in the directed sense of the contract net. Rather, a trader with an interest in participating in an under-formation transaction tenders its *bid* by interacting directly with the *market message* if invited to do so by the advertiser.

As with Sandholm's extensions, commitment in the Atomic Market is flexible. Commitment terms could change after the negotiation is underway. A no-commitment exploration could evolve into a nearly complete transaction requiring full commitment at every subsequent step. Unlike Sandholm's leveled commitment protocols, the Atomic Market does not implement levels of commitment, though *commitment* is an available *component* of the Atomic Market's modular ontology. Leveled commitment could be readily implemented provided the agent traders are capable of dealing with its implications.

The original CNP suggests a "monitoring" mode by which a manager may follow the progress of a task after it has been assigned, but before completion. Contract net managers may even control their contractors, pausing or canceling work in progress. In contrast, Atomic Market agents are equals. No provision is made for direct control of one agent by another. However, the receipt of a message does ordinarily trigger activity in the receiving agent. A small measure of "control" is thus implicit in every message.

Contract net assignees are expected to execute their tasks as delivered. By comparison, Atomic Market agents receive *market messages*, not *tasks.* A *market message* is roughly equivalent to the highest-level supertask in a contract net, and the work of a group of Atomic Market agents is to design the entire transaction in detail before executing it. So every agent is empowered to prune inefficient or undesirable components (analogous to subtasks) from the message, or to add its own terms to the message. Components may also be contingent or optional.

Contract net agents are expected to maintain lists of *possible* advertisements they might answer. That is, the contract net is task-centric. The Atomic Market, in comparison, is trader-centric. It assumes that agents (and their operators) will build relationships over time and will remember *preferred trading partners* to reduce their search overhead. This does not imply that Atomic Market agents do not search aggressively. Search is an important facet of any healthy market. During a negotiation, Atomic Market agents search part of the known space by contacting favored peers. They also search part of the unknown space – the open market – and thereby gradually cover the entire market over the course of many transactions. In the aggregate, if all traders follow this practice, the space is continuously covered by a mesh of partial searches.

As with Sandholm's extensions, and unlike the original CNP, an Atomic Market agent may make the same request of many peers, only one or some of which will receive an actual contract award. The

Atomic Market thus "fans out" transaction drafts as it explores the space of possible results. Due to this behavior, the Atomic Market is susceptible to problems of the *commons* – in particular, to a combinatorial explosion of traffic. Atomic Market agents' broad searches may render traders more vulnerable to message storms than other contract net traders.

The Atomic Market subscribes to the solutions to this problem proposed by Sandholm. In addition, a unique characteristic of Atomic Market messages encourages agents to act selectively: an Atomic Market message is protected by cryptographic signatures both over its components and over the entire message. These signatures are computationally expensive to generate and authenticate, so the market is *paced* at some rate proportional to the throughput of its traders. The Atomic Market demonstration system currently uses public key methods to generate and authenticate signatures. To truly encourage selective messaging, the signature-generation algorithm could be tweaked to make it substantially more difficult to generate than to verify, shifting the burden to senders.

Finally, the Micro-Option [26] addresses buyer-driven coordination of a complex transaction across multiple unrelated vendors, through an approach based on equities options in financial markets. The Micro-Option system considers transactions in which the smallest components are complete interactions with vendors. Much smaller component parts of such transactions are discussed in the Atomic Market. Where Micro-Options distinguish "consumer" and "resource," no equivalent distinction is made in the Atomic Market.

Axiomatic approaches

Axiomatic views of planning and coordination invoke classical AI methods that are intended to capture rational behavior. The Belief/Desire/Intention (BDI) model is the most prominent of these [27]. The BDI model seems match our intuitive understanding of how human reasoning might be modeled. BDI agents have *beliefs* (an understanding of the state of the world, including the past, from a personal point of view), *desires* (goals), and *intentions* (plans for achieving the goals). BDI architectures consider the design of individual agents capable of acting as productive group members. Although they do not explicitly implement a formal BDI model, all the agents in the demonstration Atomic Market were designed to act as individuals. The design of the Atomic Market itself was really a process of designing the space between the agents – the character of their interactions – rather than the agents themselves.

Social laws

Social approaches build on the metaphor of a *society of agents*. They ask how a system can be structured to avoid conflicts and thus avoid the need for classical planning. Such systems touch on the mechanism design work of Arrow [28]. This approach requires some form of absolute control over all agents – and requires that agents follow social laws simply because they were designed to. However, as Rosenschein observes, assuming all other agents are following social laws, an agent might do well to break the rules if it can.

The Atomic Market assumes that all agents are potentially capable of cheating, and that the agent-to-agent message protocol is therefore the proper place for a fraud prevention mechanism. Passive enforcement capabilities (the right to discard any unacceptable message) are a part of every agent. Active enforcement capabilities (reputation mechanisms) may exist in the market to improve the quality of enforcement, though they are optional.

Moses, Shoham and Tennenholtz have pursued “stable” social conventions that are suitable for individually motivated agents. [16] Their research covers the synthesis of social laws [29] and multi-agent cooperation [30].

In the tradition of this work, Walker and Wooldridge [31] explored the emergence of social conventions from within a society of agents. Given only local information, it is possible for a group of

autonomous agents to reach agreement about social conventions using only their internal knowledge, without recourse to prior global arrangements. Walker and Wooldridge formalized the process of determining the optimal function for the discovery of mutually acceptable social conventions.

Dellarocas and Klein defined a framework for "Civil Agent Societies," contract net-based agent systems that are analogous to civil human societies in that they are built on "well designed social contracts" – agreed-upon behavioral constraints – in exchange for an orderly society. [32]

Dellarocas, Klein et al have also built exception handlers for contract nets [33]. An exception handling architecture can allow a marketplace to cope with unreliable communications, limited trust among independent agents, and even systemic failures. When exception handling is built into the marketplace, agents do not need to attempt the impossible task of self-monitoring for all classes of exceptions. Exception handling *institutions* in agent societies can help guarantee efficiency and fairness for participants.

Previous experimental agent market systems

Early agent marketplaces [34] mapped traditional markets to the web, and added features that were not possible in traditional human-run, offline marketplaces.

Kasbah was a virtual marketplace similar to a classified ad system, with buyers and sellers represented by autonomous software agents. The agents were empowered to set ask/offer prices based on the *urgency* of the request, and other price-setting parameters previously provided by the agents' creators. [35]

Guttman and Maes introduced integrative negotiation [36] in which negotiation, compromise and discovery replace the traditional all-or-nothing approach of contemporary retail sales. The goal of such interactions is a more continuous market in which true preferences are revealed, not only over price, but over many features, through techniques such as multi-attribute utility theory and distributed constraint satisfaction. A related comparison of *cooperative vs. competitive* agent negotiations [37] proposes that cooperative multi-agent tools can better serve retail markets than existing competitive auctions.

MAGMA [38] was a proposed architecture for agent-based electronic commerce, that captured the elements of traditional offline markets such as communication, transfer of goods, money handling and

transaction mechanisms. The MAGMA architecture modeled traditional market activity through a platform-independent API. In contrast, the Atomic Market proposes that agent marketplaces might not be structured at all like traditional markets.

Vulkan studied the incentives of traders who could either negotiate directly with a small number of trading partners, or through a centralized market with a large number of trading partners [39]. *Efficient Mechanisms for the Supply of Services in Multi-Agent Environments* showed that direct negotiations "unraveled," leading all traders to the centralized market and destroying the profit potential of direct trade. The Atomic Market is a one-to-one architecture, but it attempts to mitigate the problems discussed in this study through a hybrid approach that encourages direct negotiations with large numbers of trading partners.

Jennings covered market-centric research systems in which agents were hosted or managed by a benevolent, limit-enforcing central authority [40]. The Atomic Market pursues a decentralized approach.

Mullen and Wellman [41] described the use of a *computational market* to help software agents allocate their limited resources when faced with a vast array of choices. This paper and the work of Wellman and his co-authors in general (e.g. [42] [43] [44]) pursued applications of economic theory in the design of multiagent systems.

In *Design methodology for secure distributed transactions in electronic commerce* [45], Portiollo et al developed a five-level framework for electronic commerce consisting of *transaction*, *transport*, *user environment*, *system management* and *business* levels.

Electronic commerce practice

The Atomic Market system stands in contrast to most commercial market-making systems. Most market systems take a conservative route, translating existing physical world Business-to-Consumer ("B2C") and Business-to-Business ("B2B") sales practices to electronic form, encoding rather than revising existing customs.

Configurable generic platforms

The emergence of an e-commerce vernacular has spawned fierce competition among providers of e-commerce platform products. IBM's WebSphere Commerce Suite[6], Oracle's E-Business Suite[7] and the CommerceOne product family[8] are all server-centric, centrally controlled systems that encapsulate "traditional" e-commerce behavior in web-based configurable systems. These systems also have many of the elements of offline management systems, such as inventory control, marketing and customer service. However, negotiation and market making are given no special consideration in these systems. The marketplaces they enable are closed, proprietary, and for use by human traders. They provide optional auction modules based on traditional auctions, but the most automation the auctions offer is proxy bidding. The CommerceOne system segregates public marketplaces, private marketplaces, auctions and procurement. Oracle provides "exchanges" but then segregates them by industry-specific offerings: transportation, marketplaces, and supply chains. The Atomic Market neither recognizes nor requires distinctions based on marketplace characteristics or industry.

Ariba

Ariba software manages a traditional trading model composed of buyers, sellers and market makers. The software attempts to carve profit niches for all three roles. Products such as Dynamic Trade support corporate procurement and provide frameworks that enable customers to configure online traditional auctions or stock exchange-like bid-ask marketplaces. Ariba addresses price setting, but does not perform sophisticated trading partner discovery or include software agents in the negotiations. Ariba is perhaps best understood as an enabler of electronic versions of traditional market models and strategies [46] [47].

---

[6] http://www-4.ibm.com/software/webservers/commerce/wcs51.html
[7] http://www.oracle.com/applications/B2B/Exchanges/
[8] http://www.commerceone.com/download/collateral/cmrc_auction_srv_brochure.pdf

Izodia (formerly InfoBank)

Like Ariba, Izodia relies on a conventional buyer-seller-market maker approach to trade. Also like Ariba, Izodia is a framework for building electronic versions of traditional marketplaces. However, in contrast to the Ariba approach of offering a selection of specialized products to be combined by customers, Izodia delivers procurement, supply and marketplace operations within a single platform. The company's position is that because every business is both a buyer *and* seller, its e-commerce platform should help manage those roles. [48]. Izodia does not discuss agent-run trading, trader discovery, or facilitation of novel market models.

VerticalNet

VerticalNet [49] builds and manages 58 specialty vertical electronic marketplaces including the Dairy Network [9] and School Buyers Online[10]. For the first five years of its existence, VerticalNet was a supplier of services – creating and managing marketplaces for customers. The company announced in April 2001 that it would begin to link its supplier network to public marketplaces and "extended enterprise networks," thereby moving toward more open markets [50]. Much of VerticalNet's prior work concerned the design and operation of private, closed marketplaces.

FreeMarkets, Inc.

Pittsburgh-based FreeMarkets, Inc. [12] pioneered reverse auctions in B2B marketplaces to help large companies cut procurement costs. FreeMarkets provides not only software but also auction design and management services. By late 2000, FreeMarkets had even conducted 50 auctions in India, valued at more than $42 million, that saved an estimated 9%[11] over non-auction prices [51]. Although FreeMarkets works closely with its customers to design auctions tailored to their industries and individual needs, this approach is not necessary ideal. Trading is not agent-driven, so traders are not compelled to define strategies or valuations before entry into the marketplace. Because bidding is not automated, traders participate in the auction as if they were together in an auction house. Partner discovery is restrained by the FreeMarkets policy of allowing only approved traders into an auction. This provides assurances about integrity but could lock out viable trading partners.

---

[9] http://www.dairynetwork.com/
[10] http://www.schoolbuyersonline.com/
[11] Savings estimate as calculated by FreeMarkets, Inc.

Smart Market

In procurement scenarios such as those offered by FreeMarkets, commitment is not equally binding on buyers and sellers. For example, auctions run by General Electric use winning bids to guide procurement decisions. However, a winning bid merely guarantees *consideration* for a contract, not the contract itself. A supplier might be driven to misstate its capacity to improve its odds, or might shy away if "cheated" after submitting a low bid that failed to generate an order. The Smart Market [52] proposes an incentive-compatible hybrid reverse auction in which the market maker advises bidders through an auction mechanism that helps bidders optimize their bids. Even with feedback and analysis services for bidders, Smart Market procurement auctions admit only monotonically decreasing bids – an auction run by a buyer still favors the needs of the buyer.

Frictionless Sourcing

Frictionless Commerce first launched a preference-based product selection system provided to buyers by sellers, then followed with a business-to-business sourcing product in February 2001 [53]. The cornerstone of both Frictionless product lines is their ability to bolster human decision-making through analysis that considers many parameters beyond price. Frictionless Sourcing is not an agent-based system per se. It accumulates and organizes information about the procurement process for human decision-makers. Frictionless has bound a reverse auction mechanism to an RFP/RFQ system, allowing purchasers to qualify vendors via RFP responses and to conduct reverse auctions to discover the best prices. The procurement system uses an approach similar to the Frictionless e-Market Suite (discussed later in this chapter), but matches offers to buyers rather than buyers to products. As with most e-markets products, Frictionless Sourcing relies on central servers for all matching and analysis.

Lost Wax

Lost Wax is an Application Service Provider (ASP) that facilitates business-to-business markets. The Lost Wax marketplace is a platform for agent-run negotiation. The Atomic Market proposes a similar approach, and was inspired in part by Prof. Nick Jennings' research into agent negotiations. Jennings is the company's Chief Scientific Officer. The Lost Wax broker-based exchange has a central "market" and a "market owner" who controls the flow of trade. In contrast, the Atomic Market has no canonical central broker. As with most electronic exchanges, XML is the basis of communications between traders and between the market and the outside world. Lost Wax agents do not appear to be able to extend the scope of the negotiation as Atomic Market agents may. Lost Wax does offer "anonymity" and others services, presumably mediated by the central broker. Finally, Lost Wax

appears to bind *traders* and *market model* to the negotiation very early in the process, so that most *discovery* is complete before the negotiation begins [54] [55]. Of all the current commercial electronic exchanges, Lost Wax is most similar to an Atomic Market-like system and the vision of peer-motivated open marketplaces.

TruExchange

TruExchange is a Lexington, MA-based B2B startup that offers a marketplace engine to create "virtual trading rooms" for diverse goods ranging from memory chips to wheat. The TruExchange interface is not unlike a trader's interface on an advanced stock-trading terminal. As with most others, the system is market-centric rather than trader-centric, relying on the company's matchmaking engine to pair buyers with sellers. An obvious concern about this system is whether it is always strategic for traders to participate in short-lived spot markets. If a critical mass of peers is not present to support a truly liquid market, sufficient "market forces" will not drive prices to a rational level. TruExchange may be appropriate for some markets in which the underlying trade already resembles the equity markets, for example, the wholesale market for memory chips [56].

Bowstreet

Bowstreet's *Business Webs* approach simplifies live Internet-based linkages between service providers and service consumers – for example, between an online computer store, a customer, a component supplier, and a credit card clearinghouse. [57] [58]. Business Webs allow site operators to *black box* real-time interactions with suppliers, freeing them from the current approach which requires learning the API of every outside supplier's system, or worse, creating and maintaining "screen scraping" applications. Bowstreet's Business Web Factory tracks XML objects via Bowstreet-specific tags that describe their functions and their relations to other objects. When a company needs to customize an application for some users, Business Web Factory dynamically revises the application, clearing out "the IT bottleneck" in the words of one large customer [59]. Web site operators can use the Bowstreet product to deploy internally defined business webs, or they may dynamically link to other Bowstreet-enabled sites.

eBay

We may view eBay as either a vertical market provider whose vertical is "general goods" or a commercial market-making system whose audience is "general traders." In either case, the popularity of eBay has had a profound impact on beliefs about the role and purpose of electronic market making.

eBay moved closer to a role as commercial market maker in Fall, 2000 when it announced it would release an API to its previously-closed trading system [60]. One driver of this decision may have been a desire to reduce the processing overhead and loss of control that resulted from uncoordinated searches by unaffiliated, uninvited, unwanted outsiders such as Bidder's Edge [61].

Agents in e-commerce

Frictionless e-Market Suite

The Frictionless Commerce e-Market Suite is a comparison-shopping engine that matches buyers with goods. Frictionless software adds feature-based comparisons within a client site or across many sites. Frictionless also runs back-end servers that consolidate and transform product and price data from many vendors into a standard form for its comparison engine.

Price comparison sites including Metaprices.com, MySimon.com and SimplyPrices.com

These sites aggregate price data derived from live lookups of a large number of suppliers. A common target supplier is online bookstores, though most of the price shopping sites offer comparisons for a range of consumer goods. Suppliers tend to be "big names" though these sites seem to have made an effort to include many suppliers in their pre-set searches. An unknown company cannot become a part of these searches purely by its own effort, because the site operators must add an interface to the supplier's data to the price-comparison engine. Unlike Frictionless Commerce, these sites aggregate information strictly by price. The better equipped of these can calculate shipping costs, and offer vendor rankings and additional product information, so that supplier selection becomes another step in the shopping process. Unfortunately, distinctive offers the vendors may make (for example, a bundle offer that provides two books for the price of one) tend to be lost in the translation to an "apples to apples" price-focused comparison system.

Auction aggregators such as BidXS.com, AuctionWatch.com

These companies also aggregate products, but execute the comparisons against auction sites rather than retailers. As with the other comparison sites, the primary motivation is price comparison.

Standards and initiatives

> *The nice thing about standards is that you have so many to choose from;*
> *furthermore if you do not like any of them, you can just wait for next year's model.*
>
> *– Andrew S. Tanenbaum*[12]

The standards discussed here cover the primitive operations of contemporary electronic commerce: transactions, agent activities, and communications. These documents evolve continuously, adding complexity and features in response to user and market demands. Such a variety of standards now exists that the choice of a subset of the popular standards for a project has become a complicated process of weighing the overhead of interoperability against the cost of noncompliance. One report concluded "The quest for information technology standards for a GII (Global Information Infrastructure)[13] is likely to remain a dicey business for the foreseeable future." [62].

Some have cautioned that international standards, especially in software development, are often formed "before the area under study has matured" [63]. Clearly some standards are necessary to facilitate substantive communications even in emerging technologies. However, these warnings deserve due consideration, especially because advanced forms of electronic commerce that do not merely mimic existing practice are not yet prevalent.

The Atomic Market explores the philosophical underpinnings for a peer-regulated agent trading society, not the practical matters of implementation. As such, it does not strive for standards compliance. Atomic Market agents adhere to a social contract whereby they agree to exchange compatible objects. Those objects could easily be conformed to any chosen standard. The standards reviewed here support much of present-day electronic commerce, and their review informs both the design of the "standards-free" Atomic Market and our understanding of the needs of agent trading programs and their operators.

---

[12] in *Computer Networks*, Prentice Hall, New York, 1996 (3rd edition)
[13] Defined as seamless worldwide interconnection of existing islands of technology and information through telecommunications, computing, and consumer electronics.

Knowledge Query and Manipulation Language (KQML) and Knowledge Interchange Format (KIF)

KIF [64] is a extended form of first order logic for encoding the content of a knowledge base. KQML is a language to support communication of KIF-like constructs among software agents [65]. The language was designed by a consortium called the ARPA Knowledge Sharing Effort (KSE) [66], a project to create infrastructures for sharing and reuse of knowledge bases and the systems around them. It is implemented not by the standards body but by independent research labs building KQML-compatible systems.

The argument remains that efforts to bind recorded knowledge to formal structures may be premature. Ginsberg [67] points out that a contemporary language could be fitted for compatibility with all existing knowledge representation (KR) methods, but that it could not possibly anticipate every form of KR developed in the future. He cautions that even mildly constraining KR standards could stifle development of non-conforming approaches. Ginsberg also suggests that research into formal KR methods should be properly labeled *research* not *standards* to preclude confusion, and that the true research focus should be "whether knowledge sharing is possible at all."

Extensible Markup Language (XML)

XML is on target to become the dominant low-level framework for automated electronic commerce. However, XML by itself is just a shell. XML-based services benefit from XML's ability to communicate and interact with complex data structures, but automated inter-operation of independent systems still depends upon explicit coordination of the data structures to be exchanged. This functionality must be created atop XML through shared schemas and data-handling processes. The appearance of competing or contradictory XML "standards" for various types of exchanges obviously confounds any approach that relies upon central coordination. XML also has an interesting single point of failure: by convention, templates and validation are provided by individual remote servers whose names are hard coded into the document. This creates tiers of vulnerability to network and server outages, a risk of overload of the XML host, unmanageable external dependencies, and potentially even the loss of data should a host go offline. The unhappy outcome of such a dependency was demonstrated in April, 2001 when Netscape removed a single file from one of its web servers. This triggered a significant distributed failure, breaking scores of individual "weblog" pages that

could not be validated or displayed without the Netscape-hosted file.[14] Atomic Market agents exchange binary Java objects rather than XML objects, for simplicity in implementation. A large-scale or commercial deployment of an Atomic Market-like system would obviously consider an XML-based approach for inter-agent messages.

Foundation for Intelligent Physical Agents (FIPA)

FIPA is a membership-based organization that builds and endorses specifications "for the interoperation of heterogeneous software agents." The FIPA repository[15] catalogs the myriad FIPA standards including agent communication, message transport and management, architectures, and applications. A complaint about FIPA may be simply that there's too much of it. Rather than considering the design of systems that adaptively learn to communicate with one another, FIPA seeks to hardwire every aspect of inter-agent activity from message structure to management.

CommerceNet and the eCo Framework

CommerceNet[16] is a global non-profit collaborative effort to build and publish standards for e-commerce. The eCo Framework, officially released in 1999, was an attempted business-to-business (B2B) XML standard connecting at least twenty other standards in an e-commerce wrapper. The eCo Framework builds a market around a central market maker. Recent news reports say eCo has "lost steam" [68]. The eCo web site[17] was down for a time but is back online with a 1999 revision date [69][70] [71].

Internet Open Trading Protocol (IOTP) RFC2801

This goal of this April, 2000 Request for Comments was to define a payment-system-independent framework for Internet commerce. Unfortunately, the protocol doesn't just address payment processing but defines documents and processes for an entire hard-wired transaction. IOTP defines rigid roles for commerce actors including "Consumer," "Merchant," "Payment Handler" and "Delivery Handler," and supports only two-party transactions. IOTP illustrates the current design ethic in electronic markets. The ad hoc, dynamic, construction-driven approach of the Atomic Market is unlike the trading environment discussed in RFC2801. [11]

---

[14] http://www.oreillynet.com/cs/weblog/view/wlg/263
[15] http://www.fipa.org/repository/index.html
[16] http://www.commerce.net/
[17] http://eco.commerce.net/

Security Services Markup Language (S2ML)

The S2ML specification was written by Bowstreet, Commerce One, Sun Microsystems, VeriSign and others with reviews by infrastructure, database and accounting firms [72]. S2ML ensures security for consumer transactions that involve many companies' web sites, via XML schemas and bi-directional messaging for authentication, authorization, and privilege-based access control. Such cross-site security management is needed to realize the multi-party trading activity carried out within the Bowstreet architecture as well as any other predominantly web-based multi-party trading platforms. The proposed standard provides bindings to protocols such as HTTP, and to messaging frameworks such as ebXML (discussed below).

DARPA Agent Markup Language (DAML) + Ontology Inference Layer (OIL)

The DAML[18] + OIL[19] [73] effort provides a semantic markup language for the description and discovery of web-based resources by software agents and other automated systems. DAML+OIL is XML-based and presents some of the problems inherent in XML, including reliance on unaffiliated third parties to host, preserve and deliver XML frameworks summoned by the DAML documents. As well, DAML is comprised of a collection of participant-submitted ontologies[20] that may be inconsistent, redundant, or contradictory.

Business Process Management Initiative (BPMI)

The purpose of the nonprofit BPMI[21] is to develop the Business Process Modeling Language, a meta-language for modeling business *practices* as XML models business *data*. BPML also provides an abstracted execution model for collaborative and transactional business processes for a transactional finite-state machine.

Electronic Business XML (ebXML)

ebXML[22] is a joint United Nations / OASIS[23] project to standardize worldwide business XML specifications, to create "a single global electronic market." Although ebXML is an open, public project, participants include BEA Systems, Bowstreet, Commerce One and many other companies.

---

[18] http://www.daml.org/
[19] http://www.ontoknowledge.org/oil/oil-faq.html
[20] http://www.daml.org/ontologies/counts/
[21] http://www.bpmi.org/
[22] http://www.ebxml.org/

EbXML is portrayed as a replacement for Electronic Data Interchange (EDI), the only previous wide-scale standard for ad hoc business-to-business transactions. EbXML is a complex layered architecture, relying upon several internal standards to tie together such services as a registry, process specifications, and messaging specifications.

Although ebXML does appear to support ad hoc discovery of services, it is also rooted in EDI, and thus not flexible as a fully ad hoc system like the Atomic Market is flexible. As with EDI, ebXML pre-declares the form of an interaction. The standard *does* anticipate security through digital signatures. It also considers privacy. But signatures are not an implicit or required element of messaging and negotiation, and privacy is provided only through the suggestion that private communications could be routed through encrypted channels. ebXML relies upon a central repository and seems to expect good conduct by the traders. For example, messages have an "isLegallyBinding" field to signal that the negotiation will require absolute commitment. However, messages are not armored against fraud in a way that allows a trader to detect changes to this or other message terms.

ebXML developers also plan to incorporate the Simple Object Access Protocol (SOAP, discussed below) in future releases of ebXML, simplifying the decision process for companies that may have had to otherwise choose between two similar but competing standards.

Universal Description, Discovery and Integration (UDDI)

The UDDI project, announced by IBM, Microsoft and Ariba in September, 2000, is a marketplace registry service. The suite of UDDI, SOAP and WDSL (announced shortly after UDDI, and discussed below) resembles ebXML. However, it is managed by for-profit entities rather than a global consensus builder such as the UN. Why not just adopt ebXML? A suggestion from the ebXML front has been that "market pressures" (read *impatience)* with ebXML standards development compelled a near-term move by businesses that could no longer wait for standards to stabilize [74]. UDDI delivers business-to-business directories indexed by name, services offered, and interface specifications, not unlike the Atomic Market registry, which could be replaced by the more robust UDDI, or vice versa.

---

[23] http://www.oasis-open.org/

Simple Object Access Protocol (SOAP)

SOAP [24] is an XML-based description of computer-to-computer message passing. It also describes optional remote procedure calls and message transmission via HTTP, the primary (and often only) bearer of SOAP messages. Messages between agents in the Atomic Market or similar systems could be encapsulated in a SOAP envelope, although they are not at present. Low level Atomic Market communications use queued, asynchronous messaging. The agents could be adapted to use SOAP's HTTP transport, though asynchronous e-mail-like implementations with external queuing seem better suited to the Atomic Market architecture, as some sort of on- or off-agent queue is inevitable. Current implementations of SOAP define only HTTP messaging, though mappings for other transports, including SMTP, are anticipated [75].

Web Service Description Language (WSDL)

WSDL describes an XML document to define how to get a web-based service, the operations supported by the service, the parameters the service expects, and the values/structures the service returns [76]. It is a product of the UDDI/SOAP working group, and is used in conjunction with SOAP to connect independent processes to remote web-based services.

With the adoption of SOAP by the W3C-sanctioned ebXML standard, it appears that UDDI/SOAP/WSDL will influence and perhaps dominate ongoing e-commerce standards development.

---

[24] http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000523

4 Atomic Market

I like to watch the Japanese in Portobello market. Some are there for the crowd, sightseeing, but others are there on specific, narrow-bandwidth, obsessional missions, hunting British military watches or Victorian corkscrews or Dinky Toys or Bakelite napkin rings. The dealers' eyes still brighten at the sight of a tight shoal of Japanese, significantly sans cameras, sweeping determinedly in with a translator in tow. A legacy from the affluent days of the bubble, perhaps, but still the Japanese are likely to buy, should they spot that one particular object of otaku desire. Not an impulse-buy, but the snapping of a trap set long ago, with great deliberation.

– William Gibson [25]

Overview of the Atomic Market

This chapter discusses the functional parts of the Atomic Market, and the behaviors of its agents. The Atomic Market assumes that all traders are represented by agents, software systems that can send and receive messages, and that negotiate on behalf of users and organizations. An agent is assumed to be the property of the person or organization it represents. Full trust in the integrity of one's agent is presumed to be reasonable and safe. The phrase *generic agent* refers to a specific, reconfigurable agent model used in the Atomic Market demonstration system to construct a variety of agents.

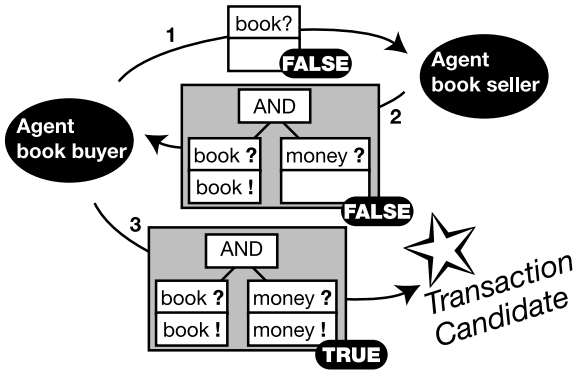


Figure 4-1: From transaction draft to transaction candidate
(the transaction draft is always carried inside a *market message*, not shown)

In the course of Atomic Market trading, agents search, negotiate, and carry out all the activities of "market participants" but strictly as peers. Seeking transactions, they hand one another proposed transactions, called *market messages*, to evaluate, modify, and possibly forward other traders. The

[25] *Modern boys and mobile girls*, The Observer, April 1, 2001

process iterates (Fig. 4-1) as a message is handed from agent to agent until it is either completed or discarded. If successfully completed, the message is called a *transaction candidate*. This model was chosen because it brings several advantages to the agents and their owners, including *flexibility*, *collaboration*, *customization* and *peer-to-peer protection*.

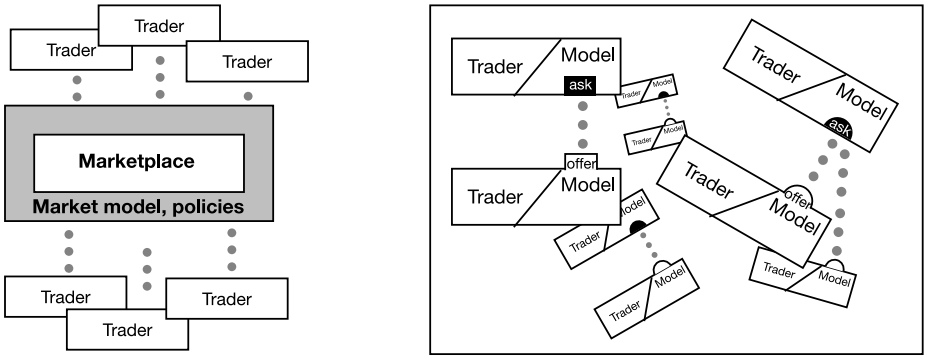


Figure 4-2: Traditional marketplace and Atomic Market

Flexibility

The continuing troubles of many Internet startups that promote unique market models, such as Mercata.com and Mobshop.com, suggest that an interesting negotiation model isn't enough to sustain a market even if the potential audience is large. A successful, sustainable market such as NASDAQ survives by maintaining a critical mass of traders who bring desirable goods to the marketplace. NASDAQ even uses its members' strength as a marketing bullet point to draw still more traders.

The failures of many early electronic merchants demonstrated that proprietary, closed marketplaces are resource-intensive, costly to run, and inflexible. What if a marketplace were completely open to all, and simultaneously reduced to a minimal framework that simply permitted traders to play out whatever strategies they chose to pursue with others? The resulting "open marketplace" (Fig. 4-2) would be free of the overhead needed to attract and keep traders. Mercata.com could aggregate the demand for *anything* in such a marketplace, from plane tickets to automobiles or even information goods, not just the small selection of hard goods that it could acquire, warehouse and sell on its own.

As an open marketplace, the Atomic Market does not attempt to disintermediate others. Rather, natural middle traders like Mercata.com and Priceline.com could represent a new kind of intermediary. Relieved of the burden of creating private marketplaces, these providers could service

other traders by offering unique *market innovations* [77] that color trader-to-trader interactions, overcoming the former segregation and inflexibility through openness and versatility.

Customization

A market message permits agents to describe their needs and offers in precise detail. Consequently, transactions may be tailored to the exact needs of individual traders. *Price* is de-emphasized as a determinant when the negotiations encompass many crucial aspects of the trade and the traded goods, allowing traders' unique capabilities to be considered.

Collaboration

> *This great increase of the quantity of work, which, in consequence of the division of labour, the same number of people are capable of performing, is owing to three different circumstances; first, to the increase of dexterity in every particular workman; secondly, to the saving of the time which is commonly lost in passing from one species of work to another; and lastly, to the invention of a great number of machines which facilitate and abridge labour, and enable one man to do the work of many.*
>
> *- Adam Smith* [26]

Like factory workers whose labor is divided, agents that collaborate with other agents gain efficiency through specialization and through the freedom of *not* having to know every required feature of every transaction, relying instead on the expertise of others.

Peer-to-peer protection

As a peer-to-peer system, the Atomic Market cannot provide global oversight – its agents must guard against manipulation by their peers. The Atomic Market's design attempts to simplify the self-protective measures any agent must undertake. It is structured so that an action taken in one trader's self-interest may benefit the entire society, as when a trader drops a forged message – preventing the message from reaching others. Some risks in the Atomic Market, and countermeasures to those risks, are discussed in the next chapter.

---

[26] *The Wealth of Nations*, [78]

The parts of a market message

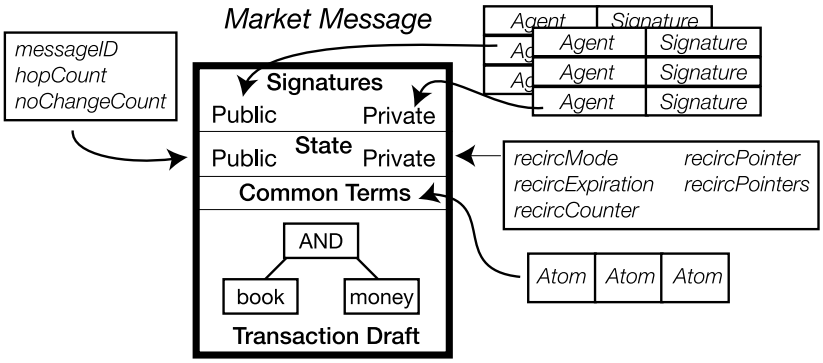


Figure 4-3: The four sections of a market message

A market message has four major sections (Fig. 4-3): *transaction draft*, *common terms*, *state* and *signatures*. The *transaction draft* and *state* sections will be briefly introduced here. Chapter 7 includes a more in-depth review of the market message object and all its sections.

Transaction draft

The transaction draft is a tree that describes the objects of exchange in a proposed transaction – goods, money, services, and contingencies, and, using the logical connectors *and*, *or* and *xor*, the relationships between them. The nodes of the tree are *components* – containers that pair individual needs with offers. Every component holds two *atoms*. An atom is the smallest object in the Atomic Market, describing in detail some negotiable part of a proposed trade. For example, an atom may hold the price of an item and the currency in which the price is specified.

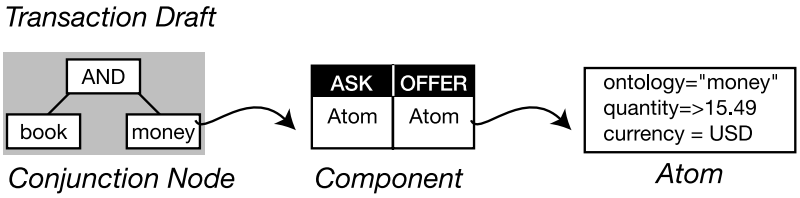


Figure 4-4: Transaction drafts, components and atoms

From right to left in Fig. 4-4, individual *atoms* are gathered into two-atom *components*. One atom in every component is an *ask* atom that represents a trader's desire, such as "I want a book." The other atom in the component should be an *offer* that complements the *ask*. The *truth value* of a component reflects of the quality of the match between the component's atoms. The truth value of the transaction draft reflects the overall viability of the transaction it defines.

State

State data within a market message improves message handling without requiring global state management. Private state data helps individual agents handle persistent messages. It is cleared when the message moves to another agent. Public state data carries metadata about the message, such as the number of hops that a message has traveled, between agents.

Truth value

Components have *truth-values* that measure *completeness*. For example, a request for a certain book that is paired with an offer of that book is *complete* or *satisfied*. A component whose requirements are satisfied is *true* while an unsatisfied component is *false.* The evaluation mechanism provides inequality-based matching ("price < 12.00") so that a request may admit many *true* answers, e.g., "a book by Ray Bradbury." In addition, fuzzy truth-values allow comparison of the relative quality of component satisfaction. When an entire transaction draft evaluates to *true*, then the traders have created a *transaction candidate* that can be executed.

Transaction candidate

In traditional commerce, *transaction* usually signifies an exchange of money for goods. In the Atomic Market, a *transaction candidate* is a multi-party document that describes many needs and offers that would be filled simultaneously if it were executed. A *transaction candidate* is a stable market message that satisfies the needs of all participating agents, is signed by all the agents, and has a *true* truth-value. Some terms may concern money and goods but others may reflect back on the transaction to specify an expiration date or decommitment fees (after Sandholm [2]). Still other terms could reference external sources such as weather reports and stock quotes. Thus, the descriptive detail of a *transaction candidate* makes it more like a contract than like a traditional transaction.

A transaction candidate may yield a conventional *transaction* via settlement acts – transaction processing, performance of promised deeds, and financial clearing. However, transaction clearing is not within the scope of this project, and the Atomic Market has no clearing mechanism. Many agent systems distribute decision processes, but engage central clearing mechanisms [79]. When a negotiation yields more than one transaction candidate, a participating agent (probably the agent that initiated the negotiation) must select the candidate that will be executed. If binding terms were present on the market message, any trader could demand execution of the described trade.

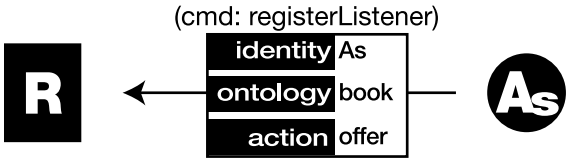
Example interaction between two agents

The following table follows the messages passed between two agents and a shared central directory called the *registry* as they construct a simple transaction. The agents meet via the registry, then trade a book for money. A complete listing and explanation of registry commands appears in *Constructor and public methods of the registry* in Chapter 7. The registry is also discussed in Chapter 8.

This example assumes the following "cold start" preconditions (and thus calls for the exchange of a larger than usual number of messages):

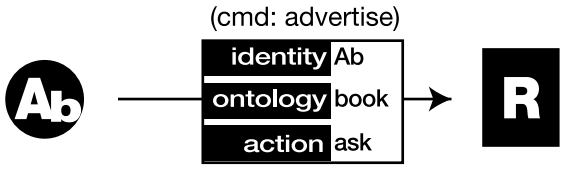
- The registry is empty
- A_b, the Agent Buyer, wants the book "Walden" by "Thoreau"
- A_s, the Agent Seller, can supply the book
- A_b and A_s have never interacted before

Many of these steps may occur asynchronously, particularly the public key exchanges.

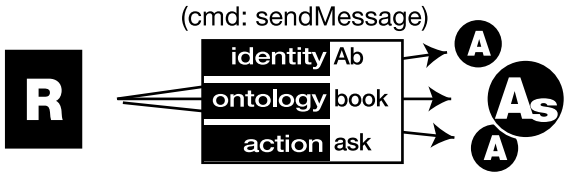


1. Bookseller adds a 'listener' to the registry
In both registry and peer communications, Atomic Market atoms are message bearers. Here, an atom holds information needed to *register a listener* that *offers books*. Once registered, agent A_s will be notified whenever a *buyer of books advertises* in the registry.

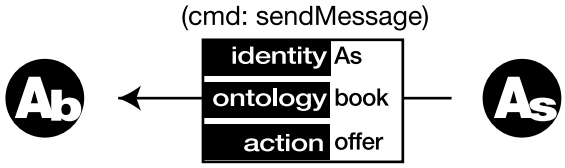
2. Bookseller queries the registry for buyers
After the seller registers for notification of the arrival of new book buyers, it issues a query to learn of already-listed buyers This is necessary because the listener registration of step 1 is only triggered by new advertisements. It does not report the status of existing registrants.



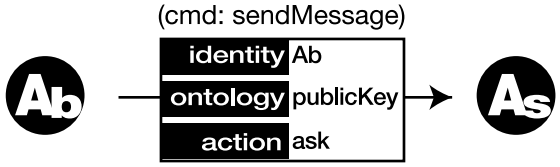
3. Buyer posts its need to the registry
Eventually a buyer agent is told to purchase a book. If the buyer doesn't know any booksellers, it advertises to the registry, triggering the registry to notify compatible listeners, as in step 4.



4. Registry notifies listeners for 'book'
The buyer's advertisement triggers event notifications to all registered booksellers. The announcement appears to have come directly from the book buyer, *not* the registry.

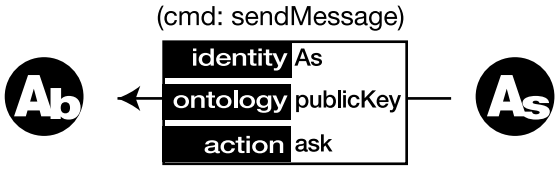


5. Seller asks buyer to consider it a potential trading partner
Upon receipt of the invitation atom "A_b, book, ask" the bookseller sends its identity to the buyer, identifying itself as a bookseller and essentially asking to be considered a trading partner.

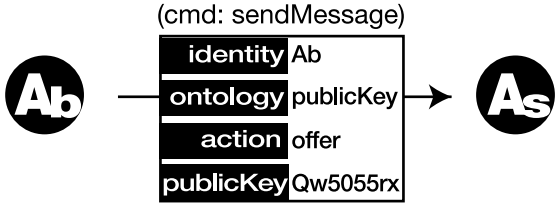


6. Buyer knows it will need seller's public key
Upon receipt of a message from an unknown agent, the buyer requests the agent's public key, to be cached for future use. Keys also travel in standard ask/offer atoms and could be bound to conditions and fees.

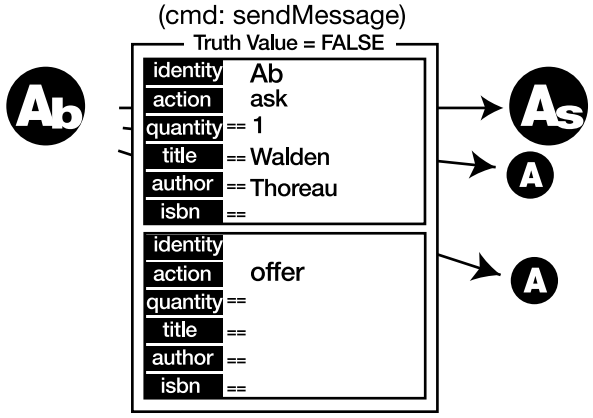
7. Seller answers buyer's key request
Upon receipt of any public key request, the seller sends its public key back to the requestor.



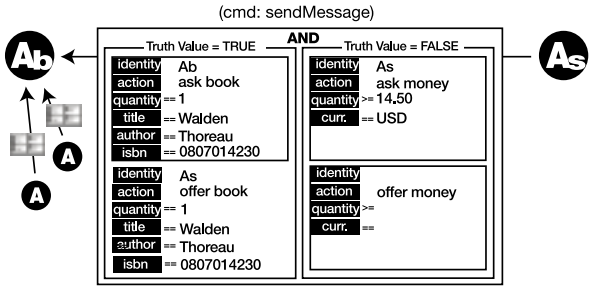
8. Seller knows it may need buyer's public key
Having heard from the buyer, the seller knows it may need the buyer's public key to process future market message, and requests it.



9. Buyer answers seller's key request
The buyer sends its public key to the seller. In the generic agents, keys are managed asynchronously in the background. A message that cannot be verified without a key is held by the agent until either the key arrives or the message expires.

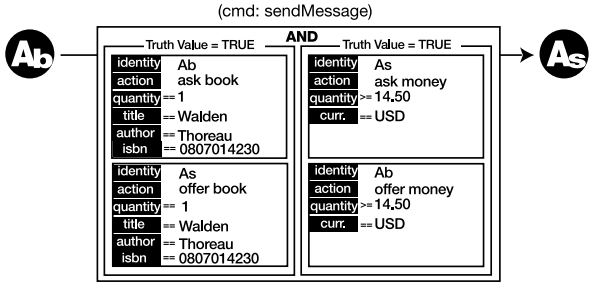


10. Buyer contacts many sellers, including 'A_s'
As solicitations arrive from eager sellers, the buyer builds a list of potential trading partners. The agent uses a stochastic method to select potential trading partners from this list. The selection is weighted in favor of known, preferred trading partners, but some newcomers are also admitted, at the discretion of the agent. The agent sends a copy of its market message – an invitation to bid – to all chosen trades.



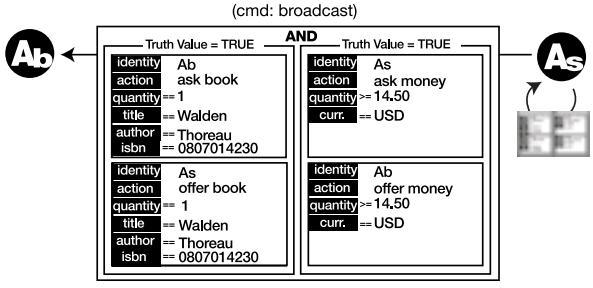
11. Some sellers alter the market message and return it

A seller that receives a market message and has an interest in bidding on it, alters the message as it sees fit, then sends it forward to another agent or back to the sender. Here, the seller A_s has filled the request for a book and added its own request for money. Other sellers also send their versions of the message back to the buyer A_b. A seller with no interest in a particular market message may discard it.



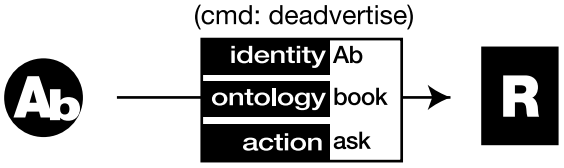
12. Buyer assesses message, adds "money" and sends revised message back to seller

The buyer may respond to many messages, depending upon commitment conditions. For example, if the message had acquired a *common term* that demanded binding agreement, the agent would have held the message until it was confident that it had found the most satisfactory result.



13. Seller broadcasts the completed *transaction candidate* to all participants

By Atomic Market rules, the agent that first recognizes a completed market message must broadcast it to all participants. A broadcast message is identified as such, so that other agents do not also rebroadcast it. If the message called for binding commitments, execution of the described transaction could be enforced by any trader.



14. Buyer removes itself from the registry

At some point, the buyer will no longer want to meet more potential trading partners. It removes itself from the registry via a "de-advertise" command that carries the agent's specific details in an atom.

Redundancy and parallel exploration

The Atomic Market implements negotiation as iterated message passing. A negotiation begins when one agent composes an initial market message and sends it to some peers. As the messages spread, some copies will be discarded, others completed and returned, and still others fanned out again by agents that initiate their own recursive searches, as in a contract net. Each agent makes the changes it believes are necessary to improve the message, and may add requirements of its own. If an agent doesn't know how to fill a need, it may send the message to an agent that it knows can provide the missing part, advertise for assistance in the common *registry*, or return the message to the sender. The broad spread of messages creates a comprehensive search of the market. Thorough searches can be advantageous to traders that search effectively, and can help keep a market competitive [80] [81].

An agent with no interest in a negotiation may discard a message. Thus, not all copies of a message are answered. The market may actually improve when some messages are discarded, because the evolutionary process weeds out disinterested traders and inferior offers, while the inherent redundancy protects the system to some extent against lost messages and broken traders.

Obviously, an agent that launches an overzealous search could easily overwhelm the system with a flood of messages flowing back toward itself. Some of the reasons an Atomic Market trader may not need to fear that its inquiries will trigger a tidal wave of messages were covered in the discussion of contract nets in Chapter 2. Additional discussion of this subject appears later in this chapter under the heading, *Traffic moderation in an unmanaged peer marketplace*.

The myriad market messages flowing from trader to trader in the Atomic Market resemble packets in a data network. The traders are similar to the network's nodes, and the characteristic behaviors, pitfalls and measures of a network can all be seen in this view. The discussion that follows will consider the Atomic Market's endorsement of, and design for, mostly-stateless agents in conjunction with stateful messages – another parallel to a data network. That discussion is followed by coverage of features of Atomic Market agent interactions that should discourage an overload of message traffic caused by excessive fan-out. As well, chapter 6 addresses the parallels between the Atomic Market and traditional networks in more detail.

Distributed state

While every agent may be engaged in many simultaneous discussions around slightly different versions of one final goal, agents are not required to track the state of every pending transaction.

Given: M: $\{M_1 \ldots M_n\}$= n possible outcomes under construction, called market messages
T: $\{T_1 \ldots T_k\}$ = k transaction candidates that emerge after coordination, $k < n$
T_f = final transaction

At any moment, the state of T_f is distributed across all participating agents who hold any subset of M. Typically a multitasking agent has some messages "on ice" awaiting replies from other agents, some message awaiting other agents' public keys for authentication, and some new, unexamined messages. As candidates for T_f emerge, they may be held briefly by an agent as it decides which to propagate and which to discard. Otherwise, an agent should not hold messages or attempt to manage the state of the entire negotiation.

Why not track everything?

Atomic Market agents generally do not track the progress of all the messages they've received or sent. At a high level, an agent needs to know if goals are being achieved, or if a change of strategy is needed. However, because message propagation is directed by independent agents outside any individual's control, a particular message may return in a very changed state, may disappear altogether, or may spawn some other message that evolves into a more viable transaction. An agent may attempt to follow the evolution of a single message to learn something about peers' strategies. However, this reveals nothing about interim changes (and thus, driving strategies) that did not involve the monitoring agent.

Message paths

Sometimes an agent must wait for information before it can process a message. For example, if an agent has advertised for booksellers via the registry, replies from interested traders will arrive sporadically, and only after some delay. The agent must store its root "I need a book" message somewhere, so that it may move on to handle other messages in the interim. Rather than keeping a separate "wait list," a generic agent just requeues a pending message to itself. The agent eventually retrieves the recirculating message and tries again to process it. If it fails, it requeues the message yet again. Incoming messages from other traders also concurrently fall into the queue. As illustrated in Fig. 4-5, messages circulate both internally and to/through others.

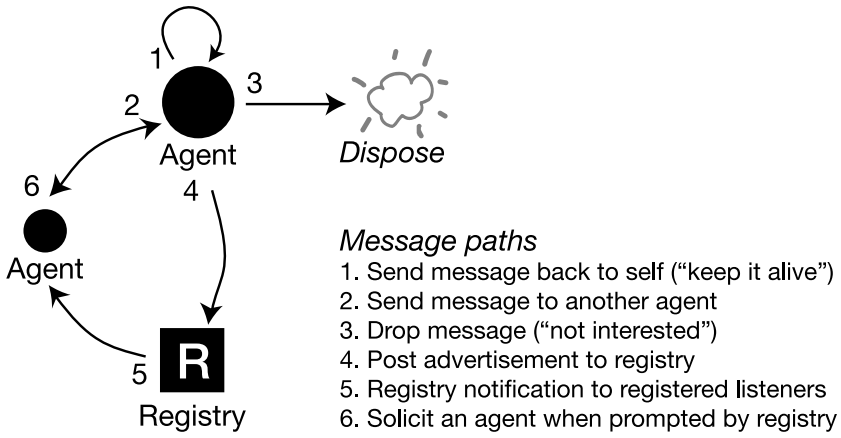


Figure 4-5: Message paths

Control of messages by stateless agents

Private state data[27] carried by a market message let the agent manage the message's distribution to others. State fields include a *recirculation mode flag*, an *expiration time stamp*, a *counter*, a *pointer* and a *bitmap*. In usage that evokes the Time To Live field of an IP packet [82], the *counter* limits the number of peers that receive a copy of a market message while the *expiration time stamp* sets an absolute stop-time. The *recirculation mode flag* signals redistribution rules for a message. *Pointer* and *bitmap* track peers that have already been sent a message, avoiding duplication.

For example, when a book-buying agent seeks booksellers, it may send the same root "I want a book" message to many potential trading partners. The agent uses the *state data* on the message to control its internal handling (Fig. 4-3). The agent sets the message's *recirculation mode flag* to `CIRCULATEUNTILEXPIRE` to signal that it should send the message to many trading partners, as they appear, not just the first. If book buyer wishes to solicit the first 100 traders that reply, or all traders that reply within five minutes of the initial advertisement, it then sets the *counter* to 100, and the *expiration time stamp* to [now + 5 minutes]. As a message recirculates, a generic agent uses the *pointer* and the *bitmap* to determine which traders to contact[28] from its list of potential trading partners, and sends copies to those traders. It decrements the counter with each transmission. When either the counter or timestamp has expired, the agent discards the message.

---

[27] See also Chapter 7: *Market Message element: State*
[28] See also Chapter 7: *Trading partner selection algorithm*

Traffic moderation in an unmanaged peer marketplace

A concern in any unmanaged network of communicating agents is that they could generate massive amounts of traffic, easily enough to destroy efficiencies and potentially enough to bring the system to a halt. In the worst case, if every agent were to begin writing to every other, N^N messages would be spawned, enough to flood the system for even a small number of agents.

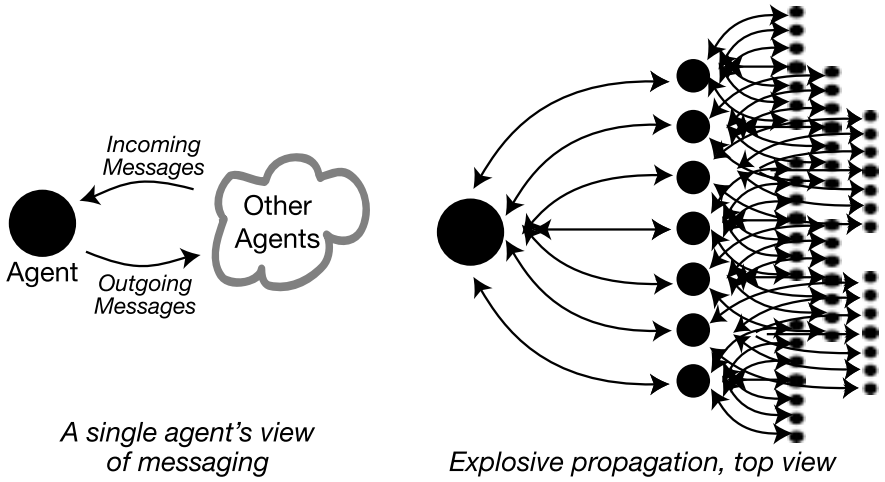


Figure 4-6: Message propagation, agent's view / top-down view of traffic explosion
An explosion of traffic can overload agents, damaging the market.

Fortunately, properly running Atomic Market agents should have no interest in writing to all others. Communications with "all others" does not lead to useful outcomes, and handicaps an agent's ability to reach *any* satisfactory end. Excluding malicious agents for the moment, an agent's goal is to complete transactions. In this framework, it can only do so through strategic and directed action. The Atomic Market architecture includes damping factors that should limit every well-behaved agent's messaging to socially tolerable levels. These factors include *feedback*, *capacity/time limits*, *directed messaging,* and *exclusion*.

Feedback

Every message sent by an agent may result in one or more messages coming back to it. An agent cannot predict the number of messages that will come back, only that the number will be proportional to the number sent. Unlike e-mail spam, Atomic Market messaging is bi-directional. Agents are discouraged from sending large numbers of messages because they may be staggered by the return traffic triggered by their own transmissions.

given: Message_In_Rate = Message_Out_Rate * return_ratio
Agent_Traffic = Message_Out_Rate + [Message_In_Rate * interest_factor]

The *return_ratio* will vary based on what happens outside the agent. The *interest_factor* measures the agent's probability of replying to an incoming message.

Capacity and time limits

Economic negotiations are often bounded by *capacity limits* and *time limits.* Most, if not all Atomic Market messages will have such limits because the shortest path to resolution is best for both an individual agent and for the agent society. In the Atomic Market, short paths are learned over time, and are sometimes discovered by explorations of randomly selected or peer-referred trading partners.

A market message is expensive to process due to the cryptographic signatures over important parts of the message. The SHA-1 algorithm imposes a mandatory computational overhead, and signatures are checked and recomputed frequently. An agent cannot estimate the validity of a given signature without testing it. This overhead is consistent as a proportion of each agent's processing power. Empirical measurements have shown that even after optimization, signature handling consumes at least 53% of every message-handling cycle within an agent. On one test system, typical message-handling cycles required 147ms to 292ms; of that, signature processing used 107ms to 247ms.

Directed messaging

Messaging is *directed* because all communications are one to one. No message explodes into many messages without specific action by an agent to disperse it. Even the *broadcast* command is implemented as a sequence of one-to-one messages.

Exclusion

Exclusion presumes a firewall-like message handler just ahead of the agent, that works with the agent to shunt undesirable messages before they reach the agent. Agents could even forestall messages from undesirables through reflective terms (e.g. "NOT any agent named n") in the market message. Through exclusion, agents that flood others will lose favor and will find their influence diminished. Because any agent may discard any message, an agent that receives a massive inflow of messages can prevent furtherance of the explosion simply by discarding most of the messages, and only processing those that seem most promising. Fig. 6-3 and the accompanying discussion consider an *ideal messaging model* for agents.

Major subsystems of an agent

Purpose

An Atomic Market *agent* is a machine that executes a private strategy upon receipt of a market message. The agent's goal is to first determine whether it cares to participate in the proposed transaction carried by the message. If so, it squares the market message with its goals, then forwards it to any other agent that needs to review it. For example, when a hypothetical *Amazon.com book-selling agent* receives a market message requesting a book, it should add a "book offered" element, then append a request for money. Finally, it should return the message to the sender to seek satisfaction of the remaining unsatisfied term: its need for money.

The major subsystems of an agent are *strategies* expressed through a *multistage resolution process*, *extension framework*, *communications, foreground process loop, trading partner selection* and *background operations.*

Multistage market message resolution

The multistage market message resolution process allows an agent to incrementally manipulate a market message based on its role and personal strategies. The five steps are *assessment of common terms*, *compliance*, *evaluation*, *strategy* and *finalization*.

> *Assessment of common terms*[29] – the agent considers whether the common terms are (1) comprehensible and (2) acceptable based on its preset or dynamically determined rules.
>
> *Compliance* – the agent conforms the market message to its needs, adding or rearranging terms. For example, the book-selling agent adds a request for money to accompany a book buyer's request for a book.
>
> *Evaluation* – the agent "fills in" as many incomplete components as possible. For example, a book selling agent fills requests for books with offers of books, and fills its own requests for money with the price of each book.

---

[29] See also Chapter 7: *Market Message element: Common Terms*

Strategy – the agent assesses the entire market message and adjusts the components in accordance with its higher-order strategies. Evaluation and strategy phases are closely linked.

Finalization –unneeded terms are stripped off a market message to remove ambiguity. For example, if a message asks for (book1 XOR book2) and a bookseller has offered both books, the message must be reduced to either (book1) or (book2).

Message resolution steps are discussed in more detail in Chapter 7.

Extension framework

The extension framework allows any agent to gain new capabilities without recompilation of the agent source. A custom class loader built into the AEXCore calls "X files," Java classes that customize agent behaviors. When properly staged and named, these are called automatically to change the agent's behavior during the five-stage resolution process just discussed.

Foreground process loop

Fig. 4-7 shows the agent's foreground message handling process. Foreground processing concerns strategic and goal-seeking activity. An agent waits for a message, acts on it, and passes the message back to itself or to trading partners, then returns to the wait state until another message arrives. Full detail of the agent's message routing algorithm appears in Chapter 7.

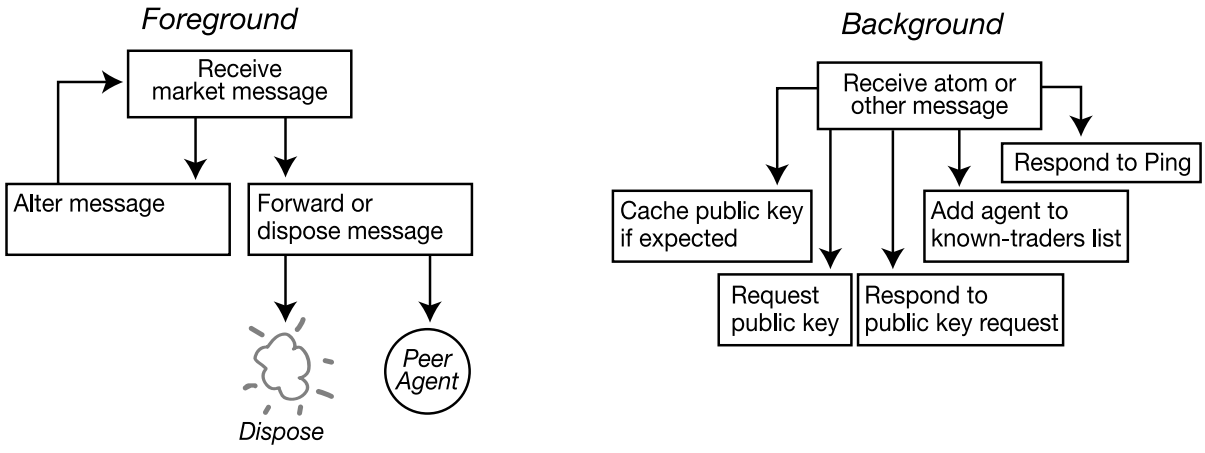


Figure 4-7: Foreground and background message handling

Background operations

Through the `Communications.start()` method and the communications thread that it launches, many background operations (Fig. 4-7) are handled without involvement of the agent's foreground process. This greatly simplifies agent design with regard to routine housekeeping chores such as key exchange, ping/pong "are you alive" messages, and tracking trading partners.

Trading partner selection

The generic agent attempts to meet new traders while hedging its bets through a heuristic method that selects from a set of known, reliable traders, and unknown traders solicited through registry advertisements. The selection is initially skewed toward known traders. As time passes or as allocation slots are consumed, other traders are randomly selected from the pool of all known possible traders. This increases the odds that new or unknown traders will have a chance to demonstrate features that make them potentially attractive trading partners. The trading partner selection algorithm is discussed in more detail in Chapter 7.

Some example agents

Some of the agents that were created for testing purposes using the generic agent as a base, and customized through rule sets and extension classes, include: *weather forecast*, *book buyer*, *bookseller* and *shipper*. Each of these is briefly discussed next.

Weather Forecast agent

Interacts with: weather_forecast ontology
Action: Retrieves weather forecast from the National Weather Service at http://weather.noaa.gov/cgi-bin/fmtbltn.pl?file=forecasts/city/(state)/(city).txt

Strategy / result: The Weather Forecast agent does not charge for its services, and adds nothing to the Market Message beyond completion of the weather_forecast component, if it is able to.

This agent uses the Xstrategy extension to run a real-time lookup of external data from the NWS web site. First, the compliance rules detect an ASK for weather_forecast. Next, evaluation adjusts all fields in the *offer* that should exactly match those in the *ask*. Next, the strategy extension fills the remaining fields with data acquired via a real-time lookup of the weather forecast site.

To request information from the weather forecast agent, create an atom with the ontology "weather_forecast". Fill the "date" field with dates of this form to conform to the national weather service format:

```
apr 23 or mon or tue or wed or thu or fri or sat or sun
```

Testing of the weather forecast agent was sometimes complicated because it worked too well. An early test case included a requirement of "tomorrow's high temperature > 70." The system worked well one Spring day, but failed the next. Springtime in Boston features 70-degree days, followed 40-degree days. The cold front that would come through the next morning was properly sensed by the agent, but caused several wasted hours of debugging perfect code to discover why transactions would suddenly not complete. A color-coded status display was immediately added, to prevent more such problems.

Book buyer agent

Interacts with: commitment, money, book ontologies

Can seek: book sellers, weather_forecast providers

Action: Tries to buy a book. Will provide money if asked to do so.

Strategy / result: Seek book providers.

This agent is proactive, and is triggered by a Catalyst to start the process of a book purchase.

Bookseller agent

Interacts with: commitment, money, shipping, book ontologies.

Can seek: shipping and shipping_insurance providers.

Action: Tries to fill requests for books. Asks for money and shipping.

Strategy / result: Answers requests that ask for a book

This agent is reactive, and is triggered by receipt of a message that asks for a book.

Shipper agent

Interacts with: commitment, shipping.

Can seek: shipping_insurance providers.

Action: Tries to fill requests for shipping, Current demo version does not ask for money or shipping_insurance, but could.

Strategy / result: Answers requests that ask for shipping.

This agent is reactive, and is triggered by receipt of a message that asks for shipping.

The Atomic Market in comparison to traditional markets

To understand the Atomic Market, it may be helpful to discuss what it is *not* by examining a traditional market. *Traditional market* refers to any manual or electronic trading center, whether it is a livestock auction or an electronic stock market such as NASDAQ. Across that continuum, we might distinguish the thing called *market* through two common properties:

> *location* – A traditional market is a *place* where compatible traders find one another. Even a completely electronic market like the NASDAQ stock market, which has no trading floor, is a *place* in this sense [83]. If two traders are not connected through NASDAQ, there is no easy way for them to trade NASDAQ shares even if they are in complete agreement about all aspects of the trade. NASDAQ recognizes the importance of *place* in framing a virtual market. It represents itself through images of real places bearing the NASDAQ mark, on large digital displays of its stocks, and by calling itself a "screen-based market." [84]
>
> *market model* – A traditional market imposes its rules of engagement on all trades and all traders. The model may be unique to the market, as with NASDAQ's decree that the market would abandon "1/16" pricing in favor of one-cent increments beginning April 2001, or Priceline.com's "Name Your Own Price™" innovation. A market may also adopt an established model, for example, an estate sale run as a "Yankee auction."

Unlike traditional markets, the Atomic Market, is more *philosophy* than *place*. There is no *place*, not even in the sense that NASDAQ is a *place*, in the Atomic Market described here. As a distributed negotiation system, the Atomic Market has no central clearinghouse like the one that anchors NASDAQ. Also unlike NASDAQ, it has neither a controlled-from-the-top *market model*, nor any model at all. Instead, the *market* exists in the moment, its state distributed among the participants in a trade. The *market model* exists in the interplay between the traders, each pursuing private goals.

The currently implemented Atomic Market does use a weak central *registry* to help agents meet one another. However, once two agents have met, they never return to the registry except to find more potential partners. Notifications of potential trading partners from the registry appear to come directly from the traders themselves, not from the registry, so the registry's presence is further diminished. A discussion of how the central registry could be eliminated, and perhaps why it should be retained, appears in the Future Work section of Chapter 7.

The Atomic Market demonstration system

Key parts of the demonstration Atomic Market include *agents*, *catalysts*, the *monitor*, and the *registry*. Common libraries include the *AEXCore*, *Rulebase*, and *generic agent*. These collections of code and class definitions simplify the creation of agents.

All demonstration agents are built atop a generic agent using the *AEXCore* and *Rulebase* libraries. Custom rule sets are used to turn generic agents into task-specific agents such as book buyers, booksellers, and shippers. The generic agent's dynamic architecture also allows the loading of custom runtime strategies. Via this architecture, more-complex procedural agents have been created, including a weather agent for forecast-based contingent terms, and a malicious agent that damages a market message to test the architecture's self-protective features.

The *Catalyst* is a "kick start" for proactive trading agents. Every agent that initiates a negotiation has its own Catalyst. In the example system, the book-buying agent's Catalyst prompts the agent to purchase a book. Once a transaction is catalyzed, the agent takes over and begins searches for, and negotiations with, trading partners.

The *Monitor* provides a top-down view of all agent interactions. A monitor is simply a receive-only agent that receives a copy of every agent status report. As implemented here, the Monitor provides a scrolling display of all agent activities, with a consolidated transcript for debugging.

The central *Registry* is implemented on a client server database. A straightforward Registry API is made available to every agent through the AEXCore.

Libraries

The AEXCore package contains message structures and manipulations that can be performed on market messages, such as creating, examining, modifying, inserting and removing components. AEXCore also provides a *Communications* class for asynchronous communications, a *Signatures* class for signature creation and validation, and a *Registry* class to abstract the Registry API. AEXCore also provides dynamic code loading to allow agent behavior to be modified at runtime.

The *Rulebase* class provides simple rule systems that agents may invoke to execute their strategies. Rulebase classes support the five-state message resolution process designed for the generic agent.

5 Risks in an Unmanaged Peer Marketplace

Doveryay, no proveryay (Trust, but verify)

– Russian saying

Traders may benefit from the freedom of the unsupervised peer-trading marketplace defined by the Atomic Market. But transactions in such an environment could prove treacherous unless adequate safeguards are provided to shield traders from malicious or negligent peers. A question that factored strongly into the design of the Atomic Market was: "how can a peer-to-peer market system with no central authority, no top-down monitoring, and no absolute enforcement provide safeguards against fraud and abuse?" Because the Atomic Market delegates both decision authority and transaction design authority to traders, they are vulnerable to new classes of attack that exploit the weaknesses of both peer systems and electronic commerce systems.

This chapter will discuss some of the threats to the traders in a peer market system, and the approaches taken in the Atomic Market architecture and demonstration system to mitigate those threats. Some risks arise only as product of intentional malice; others may result from unintended mechanical failures or flaws in agent logic.

Risk categories

The three primary classes of threats to agents are *damage to market messages*, *damage to agents*, and *deception (damage to strategies).* Atomic Market agents are not mobile, but run in a presumably safe environment. Thus, the Atomic Market does not consider the possibility of direct manipulation of an agent's logic by an opponent. Rather, it considers the effects of, and defenses against, attacks on connectivity and manipulation of market views through misleading or damaging messages.

Damage to market messages

The first class of threat concerns *damage to the integrity of market messages* and the transaction drafts they contain. The two primary risks in this class are *loss* and *fraud*. *Loss occurs* when a market message disappears in transit. *Fraud* is any case in which the transaction is changed without permission. For example, a dishonest vendor may add items to a buyer's order, or an intermediary may replace one vendor with another.

Damage to agents

The second class of threat involves *threats to agents.* These threats include *denial of service*, *misdirection*, *loops* and *deception.* In a *denial of service* attack, an agent is overloaded with unwanted traffic, hindering its ability to function. *Misdirection attacks* involve messages specifically chosen to influence an agent's internal decision processes, such as peer selection and decision making algorithms. Message *loops* signify a stalemate or failed negotiation. Agents that route messages incorrectly can also cause loops, either due to malicious intent or as the result of an error. If not detected, a single looping message can cause an avalanche of traffic resulting in denial of service. Loops can be difficult to detect, especially if more than two agents are involved.

Deception (damage to strategy)

Deception occurs when a flaw in the agent's rule set is exploited, or when a misleading message deliberately created by another causes an agent to believe it understands a message when it does not.

The steps taken in the design of the Atomic Market to mitigate the above threats fall into three categories: *passive mitigation*, *active mitigation* and *support for external recourse*.

Passive risk mitigation in the Atomic Market

The three primary passive features of the Atomic Market are *open disclosure*, *right to discard messages*, and *diversity*.

Passive features encourage traders to "play fairly." Ideally, the most an agent will be able to assume is that (1) all competitors are potentially malicious; all messages coming from the outside must be reviewed, and (2) all competitors are like itself, suspicious of others and able to detect frauds and errors.

Open disclosure

The open message format forces traders to clearly state needs and offers. All terms of the agreement must be declared plainly and unambiguously. The neutrality of the market provides no blanket assumptions – every relevant detail must be stated in transaction descriptors.

Right to discard messages

A trader may discard any message it considers undesirable, incomprehensible or ambiguous, especially if it suspects malevolence or tampering. This immediate response drives a long-term passive effect as traders eventually incorporate past results to drive future partner selection.

Diversity

The *right to discard* relies upon the diversity of messages and message paths to exploit routes around problematic traders. For example, if trader X creates market messages that are incomprehensible or fraudulent, other traders may detect and discard those messages. Trader X may simply produce messages with overly-complex conditions or difficult demands. Other market messages concerning the same transaction but not involving trader X continue to circulate. X is excluded from the transaction because its messages do not broadly propagate. Its access to future trades will also fall off as traders note it as a source of failed negotiations.

Active risk mitigation in the Atomic Market

In addition to the passive protective features of the Atomic Market, cryptographic signatures that expose unauthorized changes are an essential part of every market message. The signature system uses public key methods, permitting any agent to check any other agent's signatures. Agents that fail to check at least their own signatures risk falling into loops or becoming victims of fraud.

The Secure Hash Algorithm SHA-1 [85] creates a secure one-way hash of the components to be signed. The DSA asymmetric key standard is then used to encrypt the hash with the user's private key, creating a secure signature that may be verified by any agent using the signer's public key.

Each agent signs a market message twice: the *public signature* includes every component and common term in the entire message; the *private signature* includes only the signing agent's components.

To detect a change, an agent tests the validity of the appropriate signatures over the chosen set of components. Three cases can be discovered through combinations of components and signatures, as detailed in Fig. 5-1.

Case Detected	Components checked	Signatures checked
Message changed	All	Mine
Something about this market message has changed since the last time I handled it. The message is still evolving. Test to determine if changes are needed.		
My components changed	Mine	Mine
Components on this message that bear my personal identifier have changed since the last time I handled this message. Message is fraudulent or corrupted.		
Message complete	All	All
No traders changed this message. It is stable and requires no further negotiation. This message is a transaction candidate.		

Figure 5-1: Change and completion detection via signatures

Key pairs

Every agent generates and maintains a DSA public/private key pair. The public key is made available to any agent that requests it. The private key is a secret known only by its owner. The two keys are stored in separate files on the agent's home directory or local disk drive. Although private keys for people generally have the additional protection of a secret, unwritten passphrase that encrypts the private key, there was no benefit to creating more security over the public key files used by generic agents. The agent would have to hold some representation of the passphrase in its executable code or configuration files. That phrase could easily found out by a determined adversary. Reasonable protection options for private keys include removable hardware and careful setting of directory permissions.

Key exchange

An agent may request another agent's public key at any time, or may be called upon to provide its own public key.

Key cache

An agent maintains a cache of public keys acquired from their trading partners, to avoid retrieving keys for every interaction. The *Communications* and *Signature* subsystems can automatically maintain an agent's key cache. To guard against the insertion of bogus keys, the cache will not accept a key unless it contains an entry for that key with the placeholder value *EmptyPublicKey* where the real key should be. This token signifies that the agent is expecting the key. If there is no record for the incoming key, or if the record does not hold the *EmptyPublicKey* token, an incoming key is ignored.

Failure to sign

Should agents be permitted to omit their signatures? To the extent that a signature is a lock over an agent's own data, an agent's self-protection is its own responsibility. However, signatures serve a second role that affects the entire community. A message bearing the valid signature of an agent has the force of a contract. All agents can clearly see all signatures and can authenticate the signatures over a message. When signatures are valid, any agent participating in a transaction is empowered to hold all others accountable. In addition, if an outside authority is summoned to compel performance by an agent, the authority can independently validate a signed message without reliance on the integrity of the participants. In the early stages of a negotiation, as with contract net variants that do not require commitment until the late stages of a exchange, it is technically possible for agents to omit signatures and suffer no harm. However, the Atomic Market demonstration as implemented requires valid signatures over every exchange as a protection from fraud.

Considerations for multi-agent signatures over shared messages

All agents must agree to follow exactly the same protocol when signing market messages. An agent that does not follow the common protocol exactly will produce signatures that other traders cannot authenticate.

The system has no means of notifying traders that a public key has changed. To preclude fraud, the automatic key-exchange protocol will only accept a key from a trader if the trader receiving the key was expecting it. The protocol currently has no method that would allow a trader to "push" its new key to its trading partners. Obviously this must be dealt with before a realistic deployment would be feasible. Chapter 7 provides additional discussion of signatures and key exchange.

Support for external recourse, or, "when all else fails, call an attorney"

If traders accept that a market message, collaboratively drafted, successfully circulated and ultimately turned into a *transaction candidate,* can have the force of a contract then from time to time the things that can go wrong with paper contracts will also afflict agent-made contracts. When an agent fails to perform its agreed-upon duty, or delivers goods other than those specified in the transaction, the agents' owners may have to seek recourse through an outside party – an arbitrator or a court. One of the first acts undertaken would be the validation of signatures on the disputed transaction.

Thus it is desirable for contract enforcement that signatures be irrepudiable, that is, that a key holder whose signature over a message is valid cannot easily disclaim having signed the message. Traditionally in law, a signature can be repudiated if the signature is a forgery, or if it was obtained through fraud, undue influence (e.g. "a gun to the head") or unconscionable conduct (e.g. blackmail) [86]. Unfortunately, some legal interpretations of *nonrepudiation* with regard to electronic signatures hold, for example, that a valid digital signature "…prevents an individual or entity from denying having performed a particular action… such as… proof of obligation, intent, or commitment; or for proof of ownership" (as quoted in McCullagh). Traditionally, signatures have been made nonrepudiable through the countersignatures of third party witnesses, a function that is not present in most contemporary digital signature protocols, and that is difficult to automate.

Digital signatures are essential for electronic trade. However the risk of using such signatures will remain elevated until signature standards meet three desirable requirements for nonrepudiation: (1) that they cannot be cannot be easily removed from documents; that (2) secured, trusted message paths and (3) third-party verification can be established to secure signatures against fraud and provide impartial oversight by disinterested third parties [87].

Specific threats and their effects in the Atomic Market

Unsolicited public key

A malicious agent might attempt to replace another agent's public key with its own. The processes that accept and file public keys for the generic agent will only accept a key that the agent has specifically requested, so the window of opportunity is narrow. Once a key is on file, the agent will not replace it if another comes.

Loops

A message that circulates without ever reaching a conclusion or terminating is *looping*. Looping messages consume resources and can trigger message floods. As with looping packets in TCP/IP networks, looping Atomic Market messages may be difficult to detect, especially if the message is oscillating between two or more states that make it change slightly on every cycle. The agents most at risk for loops are always-on agents such as those representing merchants. Because of their persistent motivation to respond to incoming messages, loops through such agents could continue for long periods of time.

A loop can occur as the natural result of negotiations, or at the hands of malicious agents.

An *incidental loop* signals a stalemate in negotiations. Two or more agents are handling an incomplete market message, but the agents are failing to modify the message in a manner that acceptable to them all. It cycles with no hope of resolution.

An *accidental loop* may occur as the result of routing error by a broken or naïve agent. Each agent independently determines the next destination for a message. A broken agent may consistently choose an inappropriate trader or return every message to its sender, leading to a loop because the message has no chance of completion without assistance from a qualified agent.

An *extended loop* may occur if an agent resets either the hop counter or the no-change counter of an incomplete message without regard to the required trigger conditions, or fails to increment either counter when it should.

A *malicious loop* occurs when an agent actively tampers with messages to artificially increase traffic and trigger a *denial of service* of other agents. Atomic Market agents' exposure to this attack is

similar to the exposures of TCP/IP network nodes to the same form of attack. A malicious agent might work alone or with others to cause a cycle that never terminates. It may subtly change messages to avoid detection, or route the messages through a variety of parties to create an undetectable cycle.

Looping messages can be detected as in TCP/IP networks with cooperatively managed hop counters. Market messages have two counters: a monotonically-incremented *hop count* that indicates how many times the message has moved between agents; and a *no change count* that is incremented when an agent forwards a message without changing it, and reset whenever an agent alters the message.

Both *incidental* and *accidental loops* are detectable via the hop counters. When either counter exceeds an agent's limits, the agent may delete the message (at the risk that it may have been successfully completed after a few more hops).

When a message falls into an *extended loop*, its cycle time will be extended but is not necessarily infinite. The message will eventually reach an agent whose need for the content of the message has already been satisfied. That agent will discard the message, terminating the loop.

All loops can be eventually caught by an expiration timestamp in the looping message's *common terms* section. Agents are free to discard messages whose completion deadline has passed. If a malicious agent attempts to change the expiration stamp, it would void signatures over the stamp, and the message would be discarded as a fraud.

Finally, for protection from most classes of attack, active monitoring at local gateways may be the only solution. Traffic monitors and message firewalls that work in conjunction with the agents they protect are the ultimate flood- and attack-stopping tools.

Flooding

Loops aren't the only source of message floods, though most other floods will be the result of malice rather than accident or practice.

In a *message flood,* a malicious trader may present several pseudonyms, bidding repeatedly under different identities for business from a target trader. The malicious trader's messages overwhelm the target's ability to examine all offers, giving it a statistical edge. The target trader's view is then

skewed by the rogue. The Atomic Market has no simple protection against this attack. A defensive approach for traders to limit the effect of such an event is to maintain a private catalog of trusted trading partners. Suggestions from these agents could be combined in a controlled way with responses from unknown traders to assure that the ratio of known to unknown trading partners is maintained at a level that sufficiently exposes the true market. This approach is used in the generic agent and is supported by the Communications class.

A *stimulus-response flood* exploits an agent's automatic response mechanism to messages such as public key requests and pings. This sort of attack may be most easily stopped by firewalls and is not easily discovered except by statistical traffic management.

Diversion

In a *diversion attack*, an agent posing as a legitimate trader strips its enemies' components out of transactions and replaces them with its own. For example, a "sales-stealing agent" may pose as a shipper, and complete requests for shipping. However, the agent is actually just a middleman with a hidden gateway to shipping services. Like a password capturing "fake login screen," the agent manipulates other information in the transaction to divert business to its friends. It could replace references to BN.com with Amazon.com. A "product-stealing agent" may change a legitimate buyer's shipping address. The Atomic Market contains several defenses against diversion. The primary defense is the digital signatures over the market message. Because market messages with invalid signatures cannot become transaction candidates, an agent whose own information (such as the shipping address) has been changed would have to examine the message to correct the invalid signature, and the fraud would be detected. In the case of the sales-stealing agent, all parties might accept the diverted message and actually complete the transaction with Amazon.com instead of the intended BN.com. The question of whether such acts are ethical is unclear. We may question, but do not know, a middle agent's motives. Perhaps, if the agents are capitalists, the correct measure of success is the volume of business a trader can attract for itself by any means. Its trading partners are of course free to implement avoidance strategies.

Reverse flood

This attack is launched when one agent forges another agent's identity on a message, and sends it to thousands of others that are likely to respond. Similar attacks called *mail bombs* occur via e-mail [88]. There are no direct technological solutions to this distributed denial of service attack. However, agents that validate signatures before accepting a message are able to filter out mail bombs. E-mail

bombing victims have also adopted pragmatic solutions: detect the problem, identify the source, and pursue a resolution through civil or criminal means.

Replay

In a "replay" attack, a malicious party keeps a copy of a signed market message. Later, it re-sends the message, triggering a new transaction. The ordinary approach to protection against this attack is the placement of a timestamp or other single-use data in the record, making later re-use impossible. The *common terms* of a market message could hold a settlement timestamp or other time-limited data.

Key change

A malicious agent participates in a successful negotiation, signs the completed market message, and immediately changes its public key. When the malicious agent ultimately fails to deliver on its promises, an arbiter is engaged to settle the dispute. The arbiter retrieves the agent's public key and sees that it does not authenticate the message. There is no proof other than the word of the other agents that the first agent actually signed. Two measures to address this risk have been considered but not implemented. The first is to require agents to incorporate their public keys into the market message, thereby preserving the transaction-validating keys with the message. The second would require agents to use a trusted journalizing key server, so that a third-party record of key histories is available to settle disputes. A dispute resolution process that looks outside the immediate transaction might also discover verifiable prior instances of the agent's use of the disputed key.

Change of terms

An attacker may change the order terms subtly. For example, a seller could increase the quantity of goods, or introduce substitutions. A retailer that is out of blue shirts may substitute green, altering both its 'offer' and the buyer's 'ask' to make the transaction appear whole despite the unauthorized modification. If the agents were stateful, keeping a record of every message they handle, perhaps they could avoid this exposure to fraud through careful monitoring. However, direct monitoring of every message is inefficient and wasteful, complicating a decision process that should be focused on strategy more than self-preservation. We would prefer that a message advance toward resolution whenever it is handled. To address this risk, the generic agent does not attempt to monitor and repair messages. Rather, it relies on at-risk information to be "locked" by digital signatures, providing a direct solution to the threat by facilitating the detection of unauthorized changes.

Coordinated fraud

Several rogue traders may band together, generate a fake key pair ostensibly belonging to a targeted victim, then compose and sign a transaction that purports to include the victim as a participant. Such a fraud might be used to extract money from a victim's checking account, for example.

The self-generated, replaceable keys now used by Atomic Market traders do not offer much protection against this attack of this nature. However a bank could protect its customers by sending all messages directly to them for new signatures before processing their transactions. A better solution would be the distribution of digital certificates countersigned by a trusted third party and/or the bank itself. Properly handled certificates countersigned by the bank itself would thwart this exploit, though handling and management of certificates presents other concerns.

Interception

In an interception attack, a masquerading trader positions itself to receive messages intended for some other trader, and intercepts and responds to messages intended for its victim. In the case of the bank fraud just discussed, the bank's effort to confirm all orders by sending them directly to customers for new signatures will not thwart an interception attack. However, the additional use of countersigned digital certificates would reveal the fraud to the bank.

Bogus Stimulus

An agent could be stimulated to action by a rogue that falsified a message bearing the targeted agent's identity and containing a goal the rogue wished the target to pursue. For example, an evil author might attempt to improve sales of a slow-moving book by prompting book-buying agents with a message such as the following:

> ("id:buybot1", "action:buy", "ontology:book",
> "title: The Road Ahead", "quantity:40")

The ruse will succeed if the targeted agent *buybot1* believes this is a message of its own creation. Obviously, agents cannot blindly accept any market message as a call to action. To address this risk, Atomic Market agents always check their own signatures on messages. If an apparently self-generated signature has an invalid signature, the agent considers the message forged and discards it.

6 The Network is the Market

The network is the computer.

– Scott McNealy, CEO of Sun Microsystems

This chapter will discuss similarities between the distributed peer-to-peer Atomic Market and a distributed peer-to-peer data network such as the Internet. By adopting the language and structures of data networks, we can view the distributed e-commerce environment created by the Atomic Market from a new perspective. The chapter will briefly review the ISO model that forms the backbone of modern networks, then consider parallels between implementations of the ISO network model and the layered communications architecture of the Atomic Market. Finally, new capabilities made possible by this new view of the market will be proposed.

ISO Layer	***Example***
Application	Telnet, Netscape
Presentation	HTML, GIF
Session	SSL, HTTPS
Transport	TCP, UDP
Network	Routing
Data Link	Bits on the network
Physical	Ethernet cable

Figure 6-1: ISO 7-layer model

The ISO seven-layer network model [89] is a hierarchical architecture that makes it possible to interconnect disparate networks. Absent an architecture of this sort, every detail of communications between two computers – from the protocol to the encoding to data to the sort of network cards that are installed – would be managed by applications. Large-scale networks that permit diverse activities (such as the Internet) would be impractical or impossible.

An instantiation of the seven-layer model is sometimes called a *protocol stack.* Each layer of the stack need only communicate with those adjacent to it. Practically, this means that an "Internet application," such as a new kind of browser or a software agent, must be capable of communication with the presentation layer, but that it does not need to know anything about lower levels.

Applications never see the details of communications protocols, datagram routing, packet error correction, or the type of wiring between the computer and the network. Protocol-handling layers at the bottom of the stack do not need to consider the types of applications that may be run. To make an application "Internet ready," a programmer only needs to give the application the ability to talk to the highest layers of the stack.

In the Atomic Market, an ISO-like layered structure supports agent messaging (Fig. 6-2). At the designer's discretion, an agent's foreground process may remain happily unaware of low-level message handling and message-related bookkeeping. The Atomic Market's communications layers are not as rigidly segregated as the layers of the ISO model, and there is some cross-layer collaboration that does not follow a strict separation of roles. Primarily, these are bottom-up activities: low level communications processes update data structures that belong to the agent at the highest, or *application,* layer. For example, basic tasks such as public key exchanges, responses to "ping" messages and the tracking of known traders can be handled in the background, out of sight of the agent.

Further refinement of the layered communications structure within the *AEXCore* could facilitate plug and play agents built from a diverse selection of components.

Atomic Market communications sit atop the ISO model, and echo it:

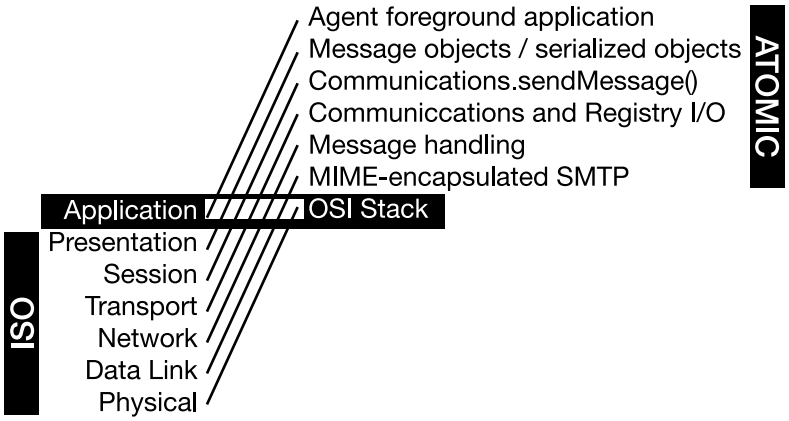


Figure 6-2: Mapping the Atomic Market to the ISO seven-layer data transmission model

ISO MODEL	**ATOMIC MARKET**
Application *Netscape, IE, Telnet, etc*	**Agent foreground application** Motivation, goals, message manipulation
Presentation HTML, GIF	**Message objects / serialized objects** Market message, ping, pong, atom, public key
Session *SSL, HTTPS*	**Communications.sendMessage()** Virtual session begins when message is first sent; continues until completion of transaction candidate or discard
Transport *Transport Control Protocol (TCP, UDP)*	**Communications and Registry I/O** Messages ←→ Bits
Network *Internet Protocol (IP)* *Routing, Physical delivery* *Address translation*	**Message Handling** Mail forwarding, firewall
Data Link *Data on the network:* *Checksum, Address, Data*	**MIME-encapsulated SMTP** (or other transport) JDBC registry connection
Physical *Cable / medium*	**OSI Stack** The OSI stack is the "physical layer" of the Atomic Market

The network market

Built in the style of a network, the Atomic Market may be expected to exhibit network properties not just at the micro-level of the network, but at a macro level visible to an observer that has a top-down perspective. Parallels between a distributed, decentralized e-commerce system such as the Atomic Market and a traditional content-agnostic data network seem intuitive when we consider a mapping from one to the other.

The Atomic Market contains agents (nodes) that exchange market messages (datagrams) with other agents (nodes) on the network. An agent processes a market message, then sends (routes) the message to its next destination. Market messages may be altered, corrupted in transit, lost, and sent into loops. There is no top-down control of the Atomic Market.

To further the analogy, simple Atomic Market agents are presumed to be entirely stateless. More sophisticated agents with motivation and satisficing properties are *mostly* stateless – retaining just enough data to make decisions in pursuit of private goals, but little more. All state data essential to the processing of a market message must be carried in the message itself, as state data are shuttled from node to node within datagrams on the Internet.

If the mapping from network to marketplace holds, then measurements and processes that work for data networks may also be applicable to the market space of the Atomic Market and other e-commerce systems. This chapter concludes with a discussion of several parallels between networks and this market, and proposes market-centric uses for these properties, drawn from lessons learned in data networking.

Parallels between data networks and the Atomic Market

At the lowest, "packet" level, we are concerned with the transmission of some bits from one node to another, or in the Atomic Market, with the transmission of a message from one agent to another. Concepts relevant to both worlds are *packet loss*, *data corruption*, *looped or cyclical routes*, and *load management*.

Packet (message) loss

In the language of the *Internet Protocol*, Atomic Market sessions are connectionless. Every market message (datagram) is self-contained. Two peers open a virtual circuit between themselves just long enough to transfer a market message. When a message returns to a previously-visited node, the node does not recall the message's last appearance, as a generic web server does not inherently recognize a user's second click at a web site as having come "from the same place" as his first.

As with IP networks, only *best effort* delivery is guaranteed in the Atomic Market. In a *packet loss* event, a market message disappears due to a faulty node, low-level communications failure, malice, or intentional disposal by an agent that is simply not interested. Because market participants are stateless and connectionless, a lost market message evaporates without detection – no alarms are sounded, no exceptions are thrown. But this is not catastrophic. In fact, if agents are working well, message loss can be healthy for the budding transaction and the traders who collaborate around it.

The great multiplicity of messages allows message loss to weed out unsuitable traders, broken nodes, and misrouted messages. In the quest to construct a final transaction candidate, hundreds or even thousands of potential messages may be spawned. If a path from origin to goal is faulty due to an uncooperative or absent agent, other paths under concurrent exploration will be discovered and the transaction may still be completed. Should all paths be exhausted without creating an executable transaction, then we conclude that at the time of inquiry there was no path from origin to goal; revision of the goal may be an appropriate response.

Data corruption

Data corruption describes a market message that has been altered in some way that makes it unrecognizable to others. In IP networks, packets are verified by checksums over their contents. In the Atomic Market, cryptographic signatures validate a market message. Signatures create a *lock* over the message, preventing undetected changes. However, signatures are much more computationally expensive than checksums, so market messages cannot move at the pace of data network messages.

Loops and cycles

An undetected and badly routed packet could circulate forever in a network. A single looping packet could trigger a cascade of traffic that eventually cripples the network.

Two approaches to prevent and correct of this problem are *routing controls* and *loop detection*. Routing controls require network knowledge and cooperation among the routing nodes. Because Atomic Market nodes do not cooperate, may work at cross purposes, and cannot be regulated from the top down, an overt control-based solution that attempts to impose routes or even routing algorithms, cannot work in every case.

The message, like an IP packet, must carry enough information to reveal that it is stuck in a loop. The *hopCount* and *noChangeCount* fields on every market message provide this information. Maintenance of these two counters is the only common obligation that requires the cooperation of most agents. Message loops in the Atomic Market actually have information value: a loop exposes a negotiation that is stalemated due to badly-matched or incomplete components.

Routing

A *route* in the Atomic Market is a solution to the original request. In other words, it is a proven path from a start state to an acceptable end state, created by the accumulation of intermediate components. For example, a plain request that says “book wanted” provides no satisfaction. To arrive at the goal state “book wanted / book offered” the book buyer must build a path that includes (at the insistence of the bookseller whose participation is essential) the components “shipping” and “money”. Thus the transaction draft that is formed through agent-to-agent negotiations is a route through one set of viable trading partners from the beginning state to the goal state.

Firewall and non-routable addresses

The ideal agent-messaging model includes a feedback-controlled firewall to block unwanted traffic, and a queue to soak up incoming messages so the agent needs to handle just one message at a time.

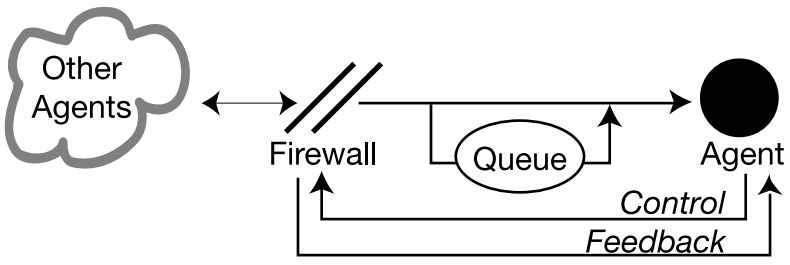


Figure 6-3: Ideal Agent Communications Structure

The firewall could also protect an agent’s identity, at least during exploratory stages of a negotiation, by mapping a temporary, externally exposed, public address to a permanent, private internal address.

This service is commonly performed on IP networks by Network Address Translation (NAT) hardware or software [90]. Because the translation is transparent to the nodes, connection through a NAT-like service for data services is quite simple. And because the alias-creating service is close to the agents it serves, it will not suffer the abuses experienced by general purpose alias services [91]. A control link from the agent to the firewall allows the agent to dynamically manage filtering and address translation.

A complication with NAT-like translation arises due to the tight security over market messages: the agent's identity is threaded throughout the message draft and is used for message delivery as well as key synchronization. If the NAT service were to change just the envelope address around the message, it would fail to hide the identity buried in the components of the transaction draft. If the translation service opened the message to change the trader's identity, it could not sign the message because it does not (and should not) hold the agent's private key. The best compromise seems to be a channel from the firewall to the agent, to pass the alias to the agent for incorporation into messages.

Broadcast

Networks provide a broadcast mode to reach all nodes in a single transmission. Because Atomic Market messages travel at the application layer and above, they do not have access to (nor do they need) a network broadcast in the traditional sense. However, the Atomic Market does define one case during which an agent needs to (1) send an identical message to several others and (2) signal that the message was delivered via broadcast. That case is the agents' behavior when a *transaction candidate* is declared. An agent that recognizes a transaction candidate invokes the *Communications.broadcast()* method to notify all traders simultaneously.

Communications.broadcast() is similar to the *sendMessage* method except that all recipients are automatically derived from the identities of traders on the message itself, and the message is tagged as a *broadcast*. The common rule for all agents, to prevent floods, is that an agent should not rebroadcast a message received via broadcast.

Using network features to probe the market

If an Atomic Market really is like a network, what can be learned from this analogy? First, network metaphors inform the design of the system and expose opportunities and vulnerabilities. Second, some network discovery and diagnostic activities can be mapped to the market.

Pinging the goal state

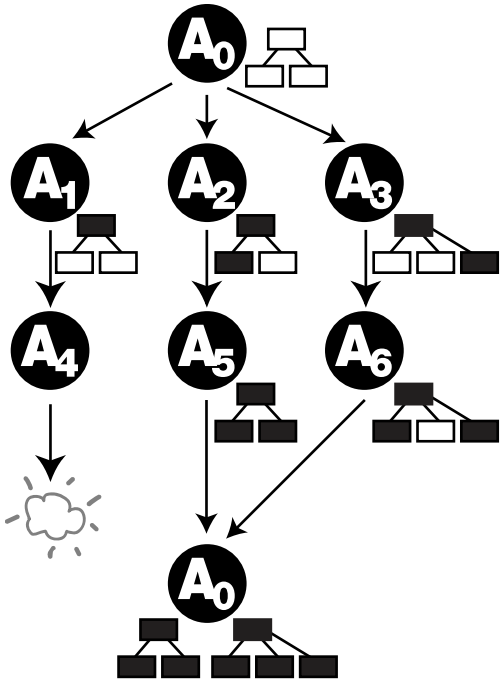


Figure 6-4: Agent A_0 "pings the goal state" by creating a complete market message (top) then sending it to nearby peers (upper middle). A_1 and A_2 supply missing information. A_3 completes one component and adds a requirement of its own. In the next iteration (lower middle) A_4 drops the message. A_5 completes it. A_6 completes a missing section but not the entire message. Finally, A_0 receives two copies of the original message – the "ping" is answered. One copy is a finished *transaction draft*. The other becomes a transaction draft when A_0 supplies a missing term.

A *ping* in a data network determines whether there is a route from a beginning node to an ending node. A *ping* in the Atomic Market determines whether a transaction draft can be defined that converts the transaction draft from the start state to the desired goal state. The *goal state* is a market message that is completed by, and satisfactory to, all participants. Its components are the *route* from start to finish.

To ping the goal state, create a market message that approximates the desired final state, and cast it into the marketplace. If the message resolves to a complete *transaction candidate*, then there is a

viable path from the initial state to the goal state. Discovery is ad hoc. The existence of the path cannot be known a priori without explicit testing. As with an IP network, we cannot determine a message's chance of successfully reaching the goal state without actually trying to reach it. The global state of the network is not visible to any agent; nor are the private and possibly changing strategies of the other agents; nor are the needs and offers of the other agents, which are not expressed until explicitly requested. An agent can only infer an unreachable goal state through a failure to achieve success after some timeout threshold has passed.

Unreachable node

A component of a *transaction draft* that cannot be satisfied (perhaps it is asking for something that is impossible to deliver) corresponds to an isolated, unreachable node on an IP network. If the component can be satisfied by some traders but not by others, paths to completion of the entire market message form around the nodes that cannot contribute to the transaction.

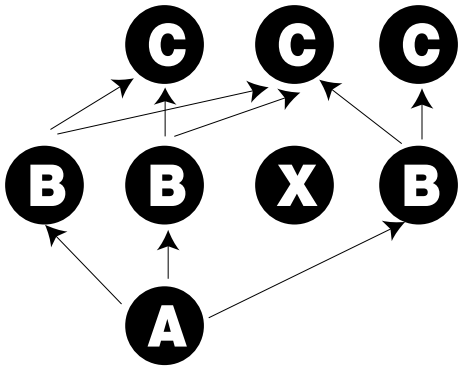


Figure 6-5: Path forms around an "unreachable node"

7 Implementation

> *ELLIE:*
> *Yes, It is our belief that the message contains instructions for building something. Some kind of machine.*
>
> *CONSTANTINE:*
> *A machine? That does what, doctor?*
>
> *ELLIE:*
> *Well, we don't know. It might be some type of advanced communication device, or it could be a teaching machine of some kind, or it might... turn out to be some kind of a transport.*
>
> – *"Contact"* [30]

This chapter will discuss the *market architecture* and reusable infrastructure elements created to facilitate the Atomic Market. It will then explain the roles and capabilities of the universal *generic agent* constructed atop that infrastructure. The chapter concludes with brief coverage of the other tools and services that support or monitor the activities of market participants.

Market Architecture

The most important component of the Atomic Market architecture is the message-passing framework and messages whose flow from agent to agent defines the market.

The fundamental job of any agent is to receive a market message, *improve* the message with its own strategies, and pass it to another trader or traders that might satisfy missing pieces. A message is alive as long as it remains in circulation; it disappears forever if dropped by any agent. Lost messages are not detected, and a trader is free to discard any message that does not interest it. All deliveries are on a "best effort" basis.

The design process for the Atomic Market concerns the content and layout of messages exchanged by agents and the definition of the methods that agents may use to operate on those messages.

---

[30] *Contact*, screenplay by Carl Sagan and Ann Druyan. Source: http://ww.scriptdude.com/

The following discussion of the Atomic Market architecture will first consider the design requirements for messages, followed by a detailed discussion of the individual elements of messages. Finally, the methods that agents use to operate on messages will be covered.

Design goals for Atomic Market messages

The market message is the heart of the Atomic Market. Every part of the system from the agent process loop to the cryptographic keys exists to support market message processing. The *market* in an Atomic Market nothing more than the state at any moment of the market messages held by all agents. A market message should be the most concise structure that supports four principles: (1) free expression of the individual terms of a transaction, including participation by more than two traders; (2) dynamic negotiation of transaction content; (3) passive support for fraud detection and mutual verification; and (4) symmetry in the control relationships between trading agents.

The market message is built from several interwoven elements, giving the message an item-by-item granularity that allows many agents to operate on a transaction while expressing their needs, offers and optional terms. Cryptographic signatures over the message and its parts protect the message from tampering.

Anatomy of a Market Message

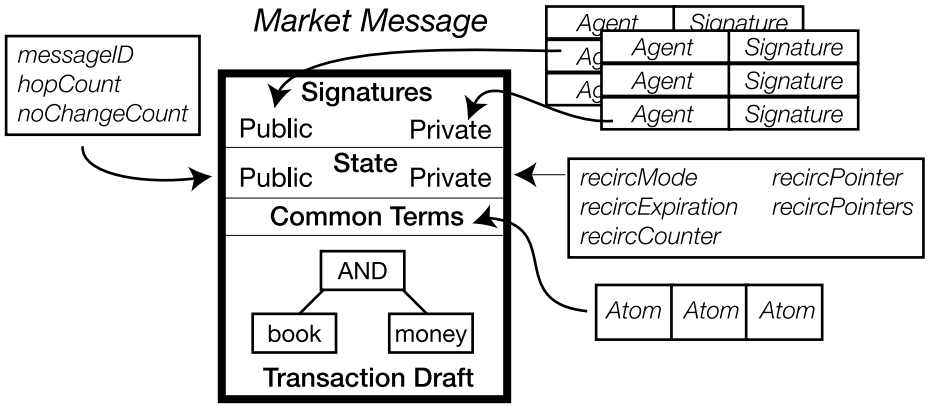


Figure 7-1: A market message and its elements

A market message contains four major sections. The sections are summarized here, then discussed in more detail.

Transaction Draft
The substance of any negotiation is located in the transaction draft of a market message. The draft contains the needs and offers of all participating traders.

Common Terms
Common terms are non-negotiable aspects of a transaction that bind all traders. For example, the declaration that an offer is valid for 30 days might appear as a common term.

State data (Public and Private)
The Atomic Market assumes agents will be mostly stateless, so essential state is carried by the market message. Private state data is used within a single agent for loop management and is reset to defaults when the message moves to another agent. Public state data persists between agents.

Signatures (Public and Private)
Cryptographic signatures protect traders against improper manipulation of market messages. Each participant creates a "public" signature over the entire message and a "private" signature over only its components. By checking its personal signature, an agent can discover that its components have been altered by another trader or corrupted in transit. The signatures also reveal when a market message is unchanging – that it is a stable transaction candidate.

Market Message element: Transaction Draft

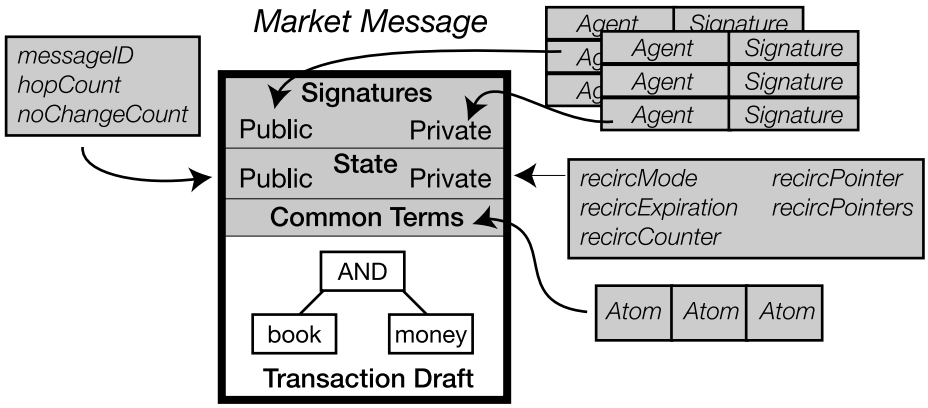


Figure 7-2: Transaction draft of a market message

A *transaction draft* is at the heart of every market message. The draft represents the substance of a negotiation between some agents – the terms of the deal, optional features, and costs. Every negotiable point of contention and eventual agreement in a negotiation session is represented in the transaction draft.

The transaction draft is a tree of individual agents' wants and offers, packaged in Atomic Market components and connected by propositional logic. For example, a request for a book, with a request for money by the bookseller, might contain this expression:

```
book( The Lorax ) AND money(USD 12.95)
```

To understand the transaction draft, it is important to first understand the elemental parts that are assembled into a draft. These parts are *atoms*, *components*, and *conjunction nodes*, which are discussed next.

Atom

The smallest objects in the Atomic Market are called *atoms*.

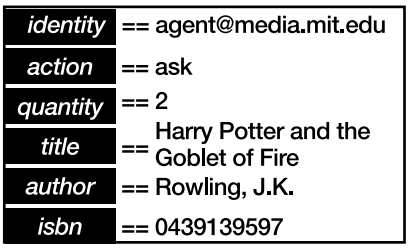


Figure 7-3: A partial atom for the ontology "book"

Every atom embodies a single domain-specific ontology appended to a common *root ontology* that is present in all atoms. An atom contains numerous *fields*. Each field inside an atom has a *field name*, a *relation*, a *data type* and a *value* that are accessible to agents. Legal data types and their permitted relations are:

Data type	Valid relations
integer	==, >, <, >=, <=, =*, !=
double	==, >, <, >=, <=, =*, !=
string	==, >, <, >=, <=, =*, !=
boolean	==, =*, !=
bytearray	==, =*, !=

The relations have their usual meanings. A new relation "=*" was contrived to represent a "don't care" case when an ontology contains fields that are not of interest for a particular transaction. For example, specifying the author of a book may be unnecessary if you know the ISBN number. Further, overly-specific field values, such as an author's name that could be written several different ways, may lead to failed negotiations as the agents haggle over irrelevant terms such as the preferred writing of the author's name.

Test cases suggested that two more-sophisticated relations, *inlist* and *between*, are needed to provide the flexibility necessary to describe common transactions. Alternatively, a LISP or Prolog engine could be substituted to operate on atoms at the lowest level. Atoms using that approach would be more complicated but very flexible in their descriptive power.

The root ontology specifies several fields that appear in every atom:

Field name	Description
%ontology	The ontology represented by this atom
%quantity	The quantity of the object or concept represented
%importance	The relative importance (used to set fuzzy truth value)
%description	A text description, for human use
%action	*offer* or *ask*
%identity	Identity of the agent that created this atom

Field names in the root ontology begin with a single "%" character. Only root ontology fields may begin with "%", to prevent name collisions with user-generated ontologies.

Atoms do not stand alone, but are handled in pairs within a *component*, discussed next. The degree to which two atoms in a component are complementary (determined by a field-by-field assessment of the atoms' relations and values) determines the *truth-value* of that component.

Component

Components, properly known as the object type `AEXComponent`, frame the smallest one-to-one interface between two agents. A full component contains one *ask atom* and one *offer atom* that should complement one other. An *ask* is matched by an *offer*, a book's title is the same for both atoms, the quantities are compatible, etc.

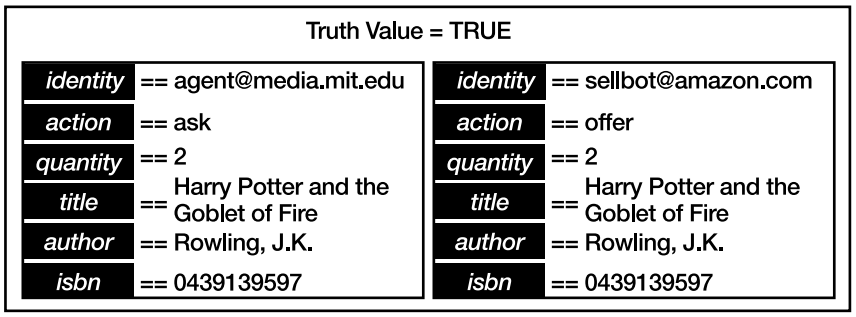


Figure 7-4: A component with two atoms

Every *component* emits a fuzzy logic truth-value that reflects the quality of the match (or failure to match) between all the fields of its two atoms. The truth-value represents the relative *trueness* or *falseness* of the component. The product of the truth-values of all components in a transaction draft is the truth-value of that transaction draft. An averaging method is used to calculate the truth-value:

> A value of `True` (the maximum fuzzy value) or `False` (the minimum fuzzy value) is assigned to the component based on the quality of the match between the component's two atoms. This result is then multiplied by the average magnitude of the two *importance* values of the component's atoms, giving a relative true or false value.

This component is false with a relative strength of 0.35:

Atom A	Atom B	Quality of match
Importance = 0.4	Importance = 1.0	n/a
Quantity = 10	Quantity < 12	True
ISBN=22322242	ISBN==12311123	False
…		

The truth-value calculation is as follows:

truth-value is determined to be False (0.0)
average *Importance* is calculated, (1.0 + 0.4) / 2 = 0.7
the range of False truth-values is 0.0 to 0.50, so (0.50 * 0.7) = 0.35

Alternate methods might attempt to reveal the degree of difference between the *ask* and *offer* atoms as a measure of the conformance of individual fields, e.g.:

Atom A	Atom B	Quality of match
price = 10	price < 12	1.0 - min[(12-10) / 12, 0] = 0.83
but…		
price = 10	price < 5	1.0 + min [(5-10) / 5, 0] = 0.0

A component by itself can only represent a single *ask* and its single matched *offer*. A more complex structure – something to glue the components together – is needed to create the descriptive *transaction drafts* employed in the Atomic Market. The mechanisms that hold components together are the *logical connectors: AND, OR* and *XOR.* The objects that hold these logical connectors and that bind the components are called *conjunction nodes*. Conjunction nodes are discussed next.

Conjunction Node

The *components* of a transaction are bound together by *conjunction node* objects. Conjunction nodes are the backbone of the transaction draft.

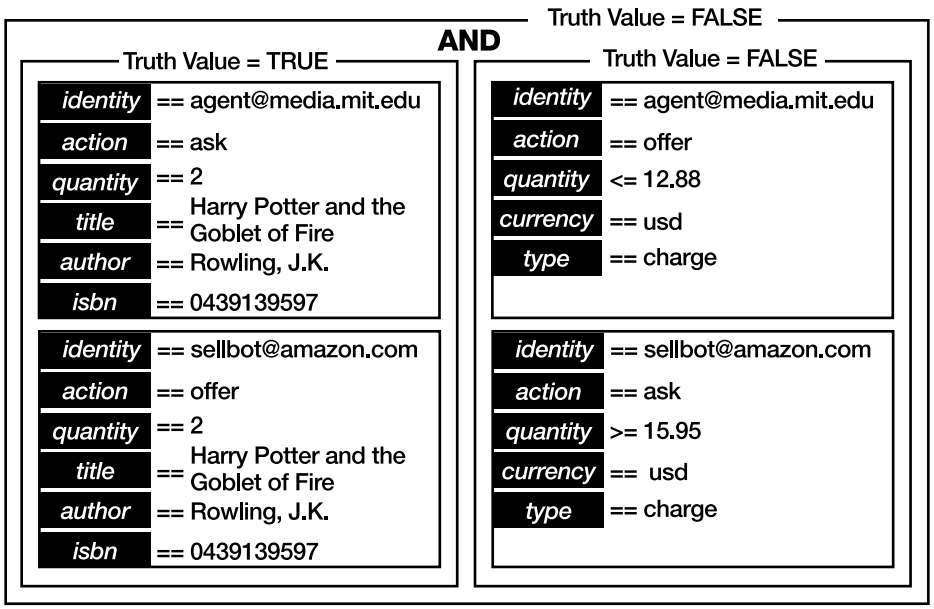


Figure 7-5: Conjunction node with two components

Because they may contain either components or other conjunction nodes, complex drafts may be created, such as a this request for one book or another book, but not both:

```
(book( The Lorax) XOR book( Horton Hears a Who )) AND money(USD 12.95)
```

A conjunction node holds a conjunction, an optional modifier, and one or more components. The conjunction node emits a truth-value that is the calculated truth-value of all the contained components when evaluated against the conjunction and modifier.

Valid conjunctions in a conjunction node are: `AND`, `OR` and `XOR`.

The only modifier available for a conjunction node is: `NOT`

Early versions of the Atomic Market restricted conjunction nodes to two components each, following the conventional notation and handling of trees. However, as the complexity of transactions increased even slightly, the drafts became unwieldy to display and handle. A

common feature of transaction drafts is that they tend to contain many `AND` conjunctions leading to tall, difficult to display trees. For example, the structures in Fig. 7-6 describe a CD purchase with requests for shipping, insurance, and payment for shipping, insurance and the CD itself.

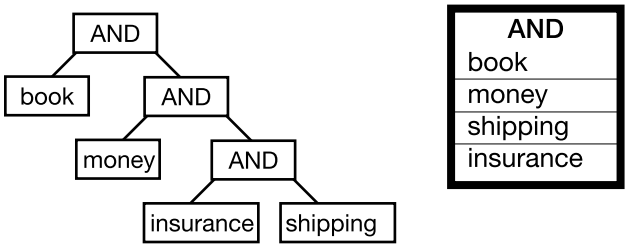


Figure 7-6: Two-component conjunction nodes (left) make tall, complicated structures
Multi-component nodes (right) are easier to handle in Atomic Market-style transactions

The conjunction node object was restructured to admit an unlimited number of components. Now, a conjunction node may state `(AND book, money, shipping, insurance)` to define a transaction that requires all four components. If many terms are ANDed, ORed or XORed together, no loss of specificity arises when all terms having a common operator are placed together. The resolution mechanism can operate on any number of terms, working pairwise until all terms have been processed. This notation and node form are somewhat unconventional, but are well suited to the needs of the Atomic Market and its traders to simplify the structure for display and analysis.[31] The component-processing logic was modified to iterate through all components on a node, rather than just one or two hard-coded elements. In practice this was not much more difficult than managing tall trees of two-component conjunction nodes.

Early forms of the conjunction node also hard coded the internal reference types. In addition to inefficient use of space (two references always went unused) the code to check for a "full" conjunction node was cumbersome and limiting. This rigidity was abandoned when the relaxed form described above was incorporated. Dynamic detection of invalid operations (e.g. an attempt to insert a market message into a conjunction) is performed using the Java *instanceof* operator so that node integrity is maintained. This change, coupled with the removal of the two-component limit provided a versatile data structure that is easy for agents to manipulate.

[31] Two-term relations are just the most basic form of this notation. Agents that do not care for the relaxed form may build two-term relations. However, there are no assurances that correspondents will follow suit.

Transaction Draft

A collection of conjunction nodes bound together (conjunction nodes may contain components *or* other conjunction nodes) is called a *transaction draft*.

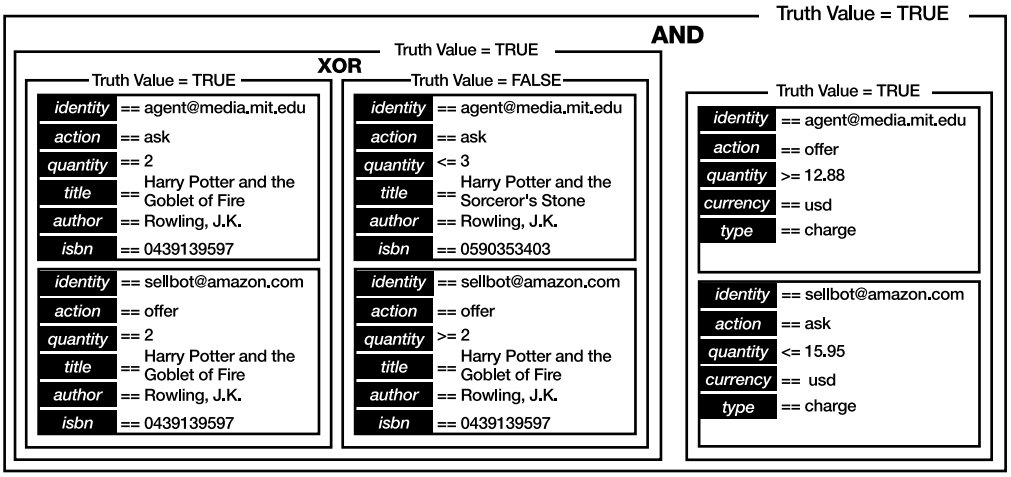


Figure 7-7: The buyer will accept either of two books

As illustrated in Fig. 7-7, the *truth-value* of the outermost conjunction node is the *truth-value* of the *transaction draft*.

Market Message element: Common Terms

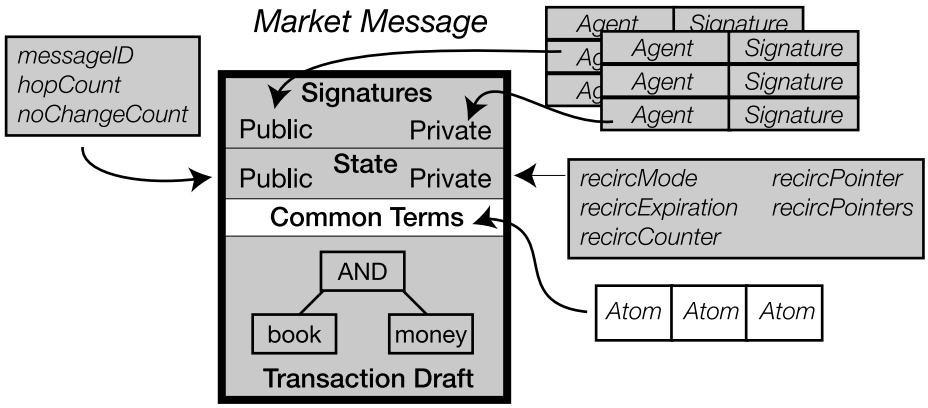


Figure 7-8: Common terms of a market message

Common terms are non-negotiable aspects of a transaction that bind all traders. Examples of expected *common terms* would be the declaration that bids will be accepted until 10:15pm EDT on June 12, 2001.

A single *common term* is simply a bare *atom* of an appropriate ontology, completely filled-out by the agent that placed it. Atoms within *common terms* are not complemented. They are declarations of required conditions rather than negotiable elements.

Any number of common terms may be attached to a market message. However, the more common terms that are present, the lower the likelihood that an agent will be willing or able to satisfy the requirements of the message. Common terms can include any atom from any ontology, but because they bind all agents and are non-negotiable, tend to include only universal constraints such as an expiration timestamp for the market message.

A future version of the Atomic Market could hold a common terms structure that is logically similar to the transaction draft – thereby making the common terms also conditional and negotiable.

Market Message element: State

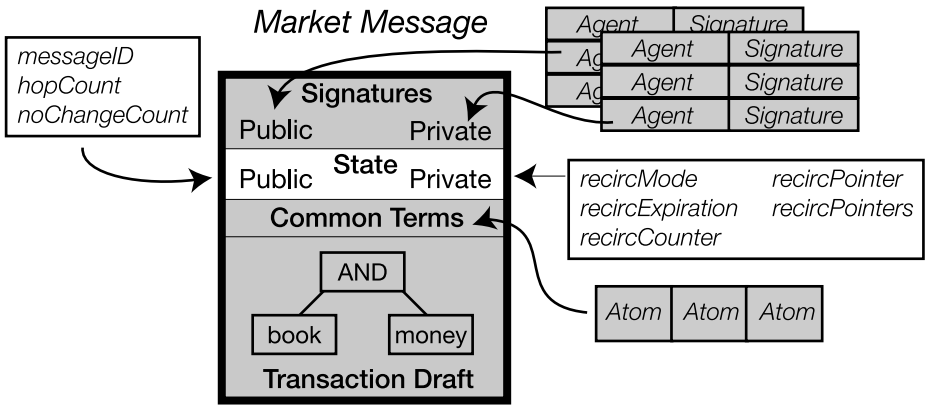


Figure 7-9: State data of a market message

The Atomic Market assumes agents will be mostly stateless. Essential state is carried by every market message in *private* and *public* sections. Private state data is used within an agent to manage a message that is recirculating locally, perhaps while waiting to meet others through the registry. Public state data persists between agents to convey information about the message.

Private state fields include a *recirculation mode flag*, an *expiration time stamp*, a *recircCounter*, a *recircPointer* and a bitmapped set called *recircPointers*. *recircCounter* limits the number of peers that receive a copy of a market message while the *expiration time stamp* sets an absolute stop-time. *recircMode* signals redistribution rules for a message. *recircPointer* and *recircPointers* are used together to track peers that have already been sent a message, avoiding duplication.

An agent seeking 20 others, or all traders for 60 seconds, would set the private fields as follows:

```
recircMode = recircUntilExpire
recircExpiration = (now) + 60000    (accept all responses for 60 seconds)
recircCounter = 20                  (or accept the first 20 responses)
```

Public state fields are the only market message elements that require cooperation by all agents. In usage that evokes the Time To Live field of an IP packet [82], *hopCount* should be decremented by every agent that handles the message. If *hopCount* reaches zero, the message has traveled too far and may be discarded. *noChangeCount* is incremented by any agent that handles a message but does not alter it. When an agent sees a message whose *noChangeCount* exceeds a personal limit, it may discard the message. If an agent changes any part of a message, it should set *noChangeCount* to zero. Optimal hop count limits are at best an “educated guess” on the part of an agent or its programmer.

Market Message element: Signatures

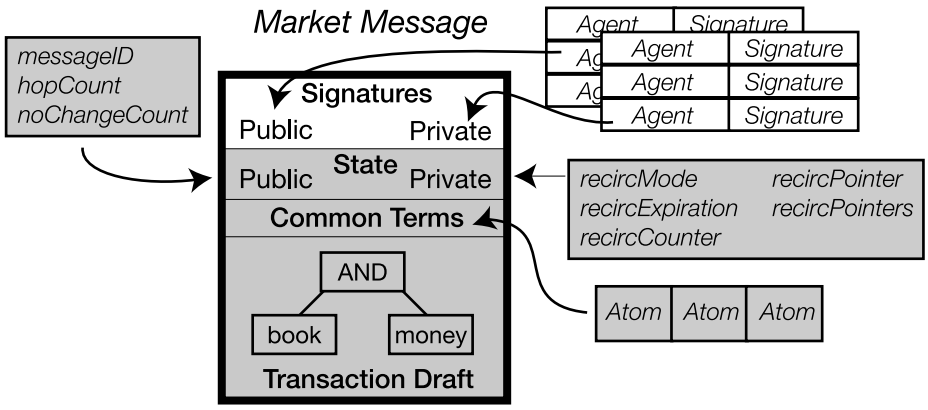


Figure 7-10: Signature block of a market message

Each agent stores two "SHA1withDSA"[32] cryptographic signatures in the market message. The "public" signature is a signed hash of the entire market message; the "private" signature signs a hash of only the agent's atoms. Any agent may test whether the message is stable by validating all agents' signatures over the message. If all the signatures are valid, then the message, if it also evaluates to *true*, is a *transaction candidate*. The agent's private signature over just its atoms serves as a protective lock – if that signature is ever invalid, the agent knows its contributions to the message have been modified without its consent.

Signature fields on the market message are supported by the `Signatures` class of the `AEXCore` package.

[32] DSA Digital Signature Algorithm (for asymmetric signing keys) with SHA-1 Secure Hashing Algorithm FIPS-180-1. SHA-1 produces a 160-bit message digest of the input data. See [92].

Constructor and public methods for atoms

Constructor

`Atom(String ontology) throws OntologyNotFoundException`
Creates an atom of type <ontology>

Methods

void set(String field, String relation, long value) throws IllegalFieldException
Sets field <field> with <relation> to <value>, e.g. "%quantity >= 3"

`void set(String field, long value) throws IllegalFieldException`
"default *set"*: sets field <field> == <value>, e.g. "identity == me@here.com"

`Object get(String field) throws IllegalFieldException`
Retrieves contents of <field> into an Object

`String getString(String field) throws IllegalFieldException`
Retrieves contents of <field> into a String

`String safeGetString(String field)`
Retrieves contents of <field> into a String. If <field> has no value, returns an empty string ("") rather than throwing IllegalFieldException

`double getDouble(String field) throws IllegalFieldException`
Retrieves contents of <field> into a double

`long getLong(String field) throws IllegalFieldException`
Retrieves contents of <field> into a long

`int getInt(String field) throws IllegalFieldException`
Retrieves contents of <field> into an integer

`boolean getBoolean(String field) throws IllegalFieldException`
Retrieves contents of <field> into a boolean

`byte[] getByteArray(String field) throws IllegalFieldException`
Retrieves contents of <field> into a byte[]

`String getRelation(String field) throws IllegalFieldException`
Retrieves relation of <field> into a String

`boolean isValidField(String field)`
Returns true if <field> is a legal field in this atom's ontology

`boolean hasValue(String field)`
Returns true if <field> has been assigned a value in this atom

`String toString()`
Returns a string representation of the contents of this atom

`void dontCareAllFields()`
makes all field relations "don't care" (=*)
this method is used for evaluating common terms

Constructor and public methods for components

Constructor

`AEXComponent(Atom a)`
Creates a component containing atom <a> in the <main> position with an empty <complement>

Methods for managing contents

`ConjunctionNode getContainingNode()`
Returns the component that contains this component, or <null> if this is the outermost component

`void addComplement(Atom a) throws IllegalComplementException`
Adds atom <a> to the component as the complement of <mainAtom>

`Atom get(String whichAtom)`
<whichAtom> should be "main" or "complement"
Retrieve the specified atom

`void insert(String _modifier, String _conjunction, ConjunctionNode _cn)`
Inserts the conjunction node <_cn> where this component is
example:
`(AND X Y) + X.insert( , XOR ,(XOR A B)) = (AND (XOR A B) X Y)`

`ConjunctionNode insert(String _modifier, String _conjunction, AEXComponent _co)`
Inserts the component <_co> where this component is (similar to the above)

`boolean isIncomplete()`
Returns <true> if either atom is missing, or if all fields are not complementary

`boolean isMine(String _id)`
Returns <true> if my identity is on either atom in this component
or
If either atom is missing or if either atom lacks an owner's identity
In other words, is this a component I *do* or *could* participate in?

`boolean isStrictlyMine(String _id)`
Returns <true> if my identity is on either atom in this component

`Atom getMine(String id)`
Return the first atom in this conjunction node that satisfies isMine(_id)

`Atom getComplementOfMine(String _id)`
Return an atom from this conjunction node that does not satisfy isMine(_id)

`double getWeight()`
Returns the weight recorded in this component's truthValue

`TruthValue getTruthValue()`
Returns this component's truthValue

Methods for traversal

void resetUnmatchedFieldEnumeration()
Resets the enumerator over unmatched fields

`String getNextUnmatchedField()`
Returns the next unmatched field in this component

Constructor and public methods for conjunction nodes

Constructors

`ConjunctionNode()`

`ConjunctionNode(String _modifier,String _conjunction,ConjunctionNode c)`
Instantiates the conjunction node and subsumes <c>
with _conjunction and _modifier

`ConjunctionNode(String _modifier,String _conjunction,AEXComponent c)`
Instantiates ConjunctionNode and populates it with <c>
setting _conjunction and _modifier as specified

Methods for managing contents

`void add(AEXComponent c)`
Adds component <c> to this conjunction node

`void add(ConjunctionNode c)`
Adds conjunction node <c> to this conjunction node

`boolean remove (AEXComponent c)`
Removes component <c> from this conjunction node

`boolean remove (ConjunctionNode c)`
Removes conjunction node <c> from this conjunction node

`void insert(String modifier, String conjunction, ConjunctionNode cn)`
Inserts conjunction node <cn> in place of this conjunction node

`ConjunctionNode insert(String modifier, String conjunction, AEXComponent c)`
Inserts component <c> in place of this conjunction node

`void setConjunction(String newConjunction)`
Sets this node's conjunction to <newConjunction>

`void setModifier(String newModifier)`
Sets this node's modifier to <newModifier>

`String getModifier()`
Returns this node's modifier

`String getConjunction()`
Returns this node's conjunction

`boolean hasContainingNode()`
: Returns <true> if this conjunction node is inside another

`ConjunctionNode getContainingNode()`
: Returns this node's parent node

`boolean has(String ontology, String action, String id)`
: Returns true if this conjunction node has an atom of <ontology> with <action> and <id>. Used by Rulesystem to determine preconditions

Methods for traversal

`void resetTraversal()`
: Resets the enumerator over this conjunction node's components

`boolean hasNext()`
: Returns <true> if there is a next component in this conjunction node

`Object getNext()`
: Returns the next component item in this conjunction node

`AEXComponent nextBreadthFirstAEXComponent()`
: Returns the next *component* in a breadth-first traversal of the current conjunction node. If a contained conjunction node is encountered, it is descended into. This method can be used to walk across an entire transaction draft.

Methods for valuations

TruthValue value()
: returns the truth-value of this conjunction node by weighing all the components' truth values and considering this node's *conjunction* and *modifier*

Constructor and public methods for market messages

Constructor

`MarketMessage(String mid)`

Instantiates a market message with ID <mid>

Methods for managing the market message

`void bind(ConjunctionNode _c)`

Attach conjunction node <c> to this Market Message
Once any conjunction node of a transaction draft tree is bound to the message, the entire tree is bound

`String getID()`

Return the ID of this market message

`int getNoChangeCount()`

Return the no-change counter of this market message
How many hops has it traveled without a revision?

`int getHopCount()`

Return the hop counter of this market message
How many hops has it traveled?

Traversal methods – general information
All the traversal methods are designed for use as elements of *for* loops. They return <null> when there are no more elements to be traversed.
Example:

```
for (Object o = this.nextBreadthFirstAEXComponent();
       o != null;
       o = this.nextBreadthFirstAEXComponent()) {
              do stuff here
}
```

Traversal methods over the Common Terms

`void resetCommonTermsTraversal()`

Resets the traversal pointer over the common terms

`boolean hasNextCommonTerm()`

Returns <true> if there is a next common term

`Atom nextCommonTerm()`
Returns the next common term

`Atom nextCommonTerm(String id)`
Returns the next common term created by agent <id>

`void addCommonTerm(Atom a)`
Adds <a> to the message's common terms

`void deleteCommonTerm(Atom a)`
Deletes <a> from the message's common terms

Traversal methods over the transaction draft

`void resetTraversal()`
Resets the traversal pointer over the transaction draft

AEXComponent nextBreadthFirstAEXComponent()
Returns the next component in a breadth-first traversal of the transaction draft

`AEXComponent nextBreadthFirstAEXComponent(String id)`
Returns the next component that satisfies `AEXComponent.isMine(id)` while stepping breadth first across the transaction draft. In other words, it returns the next component that is potentially of interest to <id> because it either bears <id>'s identity or has an opening that <id> might attempt to fill.

`AEXComponent nextStrictBreadthFirstAEXComponent(String id)`
Similar to the above, but tests with AEXComponent.isStrictlyMine(id) to return only those components explicitly bear <id>'s mark

`AEXComponent nextUnmatchedBreadthFirstAEXComponent()`
Return the next component that has an unmatched field (that is, the *ask* and *offer* atoms are not complementary)

`AEXComponent nextUnmatchedBreadthFirstAEXComponent(String _id)`
Return the next component that has an unmatched field (that is, the *ask* and *offer* atoms are not complementary) that satisfies AEXComponent.isMine(id)

`ConjunctionNode nextBreadthFirstConjunctionNode() {`
Return the next conjunction node

`ConjunctionNode nextBreadthFirstConjunctionNode(String _id)`

Return the next conjunction node containing a component that satisfies AEXComponent.isMine(id)

`Object nextPseudoRPNElement() {`

Returns a pseudo-RPN representation of this transaction draft, one object or operator at a time. This is called "pseudo RPN" because it uses parentheses to signal bounds around the arguments for each operator due to the unlimited number of components that might be contained in any conjunction node.

Example output:

```
(AND book money shipping)
```

`Object nextTrueRPNElement() {`

Returns a true RPN representation for the tree that is the transaction draft. Repeats a conjunction as necessary to conform to RPN stack behavior.

`String getRPNString()`

Returns a string representation of the RPN version of this transaction draft.

Signature methods

`boolean myComponentsUnchanged(String id,Signatures s) throws AEXSignatureException`

Returns <true> if the public key in <s> for <id> validates the stored signature over the components that have identity <id>. An agent uses this method to learn if its own components have been altered while the message was traveling.

`boolean isUnchanged(Signatures s) throws AEXSignatureException`

Returns <true> if all signatures over the market message are valid. If so, the message is stable – it is stalemated and incomplete, or successfully finished.

`void secureMarketMessage(String id,Signatures s) throws AEXSignatureException`

Calculates the agent's signatures and stores them in the signature block.

`boolean isComplete(Signatures s)`

Returns <true> if all components are properly complemented

`Vector getIncompleteComponents()`

Returns a vector containing all the transaction draft's incomplete components

`AEXComponent getFirstIncompleteComponent()`

Returns the first incomplete component from the transaction draft

`Vector getInvalidSignatureIDs(Signatures s) throws AEXSignatureException`
Returns a vector of all agents whose signatures are not valid

`String getFirstInvalidSignatureID(Signatures s) throws AEXSignatureException`
Returns the ID of the first agent whose signature is not valid

`boolean allPublicKeysAreCached(Signatures s)`
Returns <true> if the agent has all the keys it needs to validate this message

`Vector getMissingPublicKeys(Signatures s)`
Returns a vector of all agents whose keys are needed to validate this message

The AEXCore

AEXCore is a collection of classes and methods that make it easy to build agents for the Atomic Market. AEXCore encapsulates all the data structures and message-manipulating methods discussed previously, plus policy frameworks like digital signature procedures. Using these methods and data structures, individual agents can pursue the strategies dictated by their individual programming, while maintaining compatibility with other traders.

AEXCore encapsulates the following major categories:

Atomic Market logic and data structures
Atoms, Components, Conjunction Nodes and Market Messages
Fuzzy logic support and evaluation

Market operations
Digital signatures and key management
Communications
Registry

AEXCore is bound into the package `edu.mit.media.jim.aexcore` and should be linked into any Atomic Market agent and imported into the source:

```
import edu.mit.media.jim.aexcore.*;
```

The Atomic Market logic and data structures features of the AEXCore have just been covered. The next section will discuss the three market operations features of the AEXCore: *signatures*, *communications*, and the *registry.*

AEXCore: Digital signatures and key management

The s*ignature* fields on the market message are supported by the `Signatures` class, which provides key generation, key management, and digital signature services. In conjunction with the AEXCore `Communications` class, the `Signatures` class also provides automatic key exchange and key caching. The class uses the Sun Java Security API `java.security` [93].

Signatures are "SHA1withDSA"[33] hashed cryptographic signatures. To check the signatures created by other agents, and agent needs their public keys. The `Signatures` class has key exchange protocols to handle this task, transparently exchanging keys via the `Communications` class.

[33] DSA Digital Signature Algorithm (for asymmetric signing keys) with SHA-1 Secure Hashing Algorithm FIPS-180-1. SHA-1 produces a 160-bit message digest of the input data. See [92].

Elements of the signature subsystem

Personal Key Pairs

Every agent maintains a key pair consisting of a *public key* and a *private key*. The keys are stored locally in two separate files.

Personal Key Cache

Every agent maintains a cache of the public keys of its trading partners. At the agent's discretion, the `Communications` and `Signature` subsystems will automatically maintain the key cache. `Communications` will automatically request a key from any agent that interacts with the local agent. When an agent needs a trader's public key to validate a market message, it holds the message until the key arrives from the other trader or the message times out.

The key cache will not accept a new key unless the agent has created slot for the incoming key, containing a special constant token `EmptyPublicKey` in the place of a real key. This signals that the agent is expecting the key. If there is no record for the incoming key, or if the record does not hold the `EmptyPublicKey` token, an incoming key is ignored.

Key exchange

To request a public key from another agent:

Place an `emptyPublicKey` in the key cache in the other agent's name

Instantiate an atom of ontology `publickey`

Fill in your identity, and make the `action = ask`

Send the atom to the agent.

Cooperation / standardization of signatures

The SHA-1 hash employed in market message signatures is extremely sensitive to slight variances in input data. Therefore, all agents must follow exactly the same protocol when signing market messages. A standard method is provided to all agents through the `AEXCore` method `secureMarketMessage`. This method traverses all components depth-first, gathering a byte array of each and merging it into the accumulated signature. If the terms of a market message are rearranged, even if their meaning remains unchanged, the signatures will not be valid.

A vulnerability of this approach is that the fields of a component might not be read in a repeatable order. The `Hashtable` that frames the basic structure of the `atoms` does not guarantee that an

traversal of its contents will follow the same sequence every time. Because field names must be unique within an atom, a sorted data structure such as SortedMap would address this problem. Difficulties with the implementation of `SortedMap` across platforms, specifically on the Macintosh Java 1.1.8 release with Sun's Collections for Java 1.x, precluded implementation of this solution in the current `AEXCore`. Fortuitously, the iterators of `HashTable` do return the fields in a consistently repeatable order. However, a more reliable solution should be implemented as soon as practicable.

Key revocation and replacement

The system presently has no means of notifying traders that a trader's public key has changed. There is no method to allow a trader to "push" its new key to its trading partners. Obviously this must be addressed before a distributed deployment is feasible.

Possible approaches:

- If a trading partner's signature does not authenticate after a round-trip exchange with that partner, request the partner's key a maximum of 1 time.

- Permit the background processor to process a key-offer by turning it into a key-request. This opens a vulnerability to fraud.

- Use public key servers and respect public key revocation notices – this is perhaps the most robust solution.

Constructor and public methods for signatures and keys

```
Signatures(    String publicKeyFile, String privateKeyFile, String keyCacheFile,
               String id) throws AEXSignatureException
```
Reads public key, private keys and key cache from files

```
Signatures(String publicKeyFile, String privateKeyFile) throws AEXSignatureException {
```
Alternative constructor for use when there is no key cache

```
void makeKeyPair() throws NoSuchAlgorithmException
```
Generates a new Diffie-Hellman public/private key pair.

```
byte[] sign(byte[] data)
throws NoSuchAlgorithmException, InvalidKeyException, SignatureException
```
Sign `data` with my private key, and return the signature as a byte array

```
boolean validate(byte[] assertedSignature, byte[] data) throws AEXSignatureException
```
Return <true> if the `assertedSignature` authenticates `data`

```
byte[] getPublicKey() throws AEXSignatureException
```
Return my public key as a byte array

```
boolean anyKeyInCache(String id)
```
Is there any entry for <id> in the public key cache?

```
boolean realKeyInCache(String id)
```
Is a valid key (not an emptyPublicKey) in the key cache for <id>?

```
boolean emptyKeyInCache(String id)
```
Is an emptyPublicKey in the key cache for <id>?

```
byte[] getCachedPublicKey(String id)
```
Get the public key for <id> from the key cache

```
void cachePublicKey(String id,byte[] newkey)
```
Add the public key <newkey> to the key cache for <id>

AEXCore: Communications

Agents communicate by sending objects to one another through the `Communications` class of the AEXCore. Through the `Communications.start()` method and the communications thread that it launches, many background operations are handled at the communications layer without involvement by the Agent's foreground process.

Communications services for agents provide:

- Transfer of any object to or from any other agent
- Registry-to-agent notifications
- Background *bookkeeping* such as key exchanges and registry notifications
- Disposal of messages that exceed agent-specified limits

The communications system abstracts all messaging so that the underlying communication process may be changed without changes to the agents themselves.

All objects communicated in the Atomic Market are handled both internally (inside an agent) and externally (when communicated between agents) in their native form as binary Java objects. They could be transmitted in a neutral form such as XML, and processed internally as such. The decision to handle all objects in native binary format was reached in consideration of constraints on this project, and to maintain the research focus on protocols for message handling and collaboration, rather than cross-platform communications. Any platform-neutral language could describe all architectural elements of the Atomic Market.

Agents are free to send *any* objects to one another. However, because Atomic Market agents only recognize the predefined message formats, most will discard any messages not drawn from the set of expected objects.

The collection of communication objects evolved as the project was constructed. The challenge in designing these objects and specifying their roles was to accommodate new needs without adding complexity, and to reuse existing types when appropriate. In most cases, this process yielded a more simple architecture.

Communication objects

Five objects are recognized as valid communications between agents in the current version of the Atomic Market: *market message*, *atom*, *ping*, *pong* and *object*.

MarketMessage

The most common object exchanged by agents in the Atomic Market is a *market message* built collaboratively by two or more agents.

When an agent receives a market message, it should evaluate the message to determine whether (1) the message is of interest; (2) the message is acceptable as is, or requires modification

If the agent's strategies dictate that the agent should not participate, the agent may (1) discard the message, (2) return it to the sender unchanged, or (3) send it to a third party. These options reflect the breadth of agent autonomy in the Atomic Market – no agent can control what another does with a message.

If the agent will participate in the transaction, it should inspect the message for compatibility with its goals. If the agent likes the message as it stands, it signs the message, and passes it to the next trader. If the message should be *improved* in that agent's view, the agent alters the message, signs it, and pass it to the next trader.

Atom

An atom is the simplest object exchanged by agents. Bare atoms are sent between agents for signaling, as follows:

> (1) to seek compatible traders: an agent in search of a compatible trading partner prepares an atom containing its needs or offers, and posts it to the registry.
>
> (2) to respond to a compatible trader's advertisement: upon discovery of a possible trading partner, an agent prepares an atom containing its needs or offers, and sends it to that trader. An agent that receives such an atom should determine whether it is interested, and if so, should reply with a market message, initiating a negotiation.

(3) to exchange other message fragments: the only fragment now in common use is a public key carried by an atom of ontology "PublicKey," created expressly for this purpose. Public keys are not ordinarily bound into market messages, so their exchange via plain atoms is appropriate. Further, this exchange is compatible with the ordinary behavior of agents when responding to other bare atoms as described in (1) and (2).

Ping and Pong

While testing communications, *ping* and *pong* objects were created to help debug the `Communications` class. *Ping* and *pong* could have been built from regular atoms just by creating two new ontologies, but were made unique objects to allow them to be recognized at the lowest layers of the `Communications` class. These objects work just like *ping* and *pong* packets in TCP/IP or IRC – when a *ping* is heard, the receiving agent may answer with a *pong. Ping* and *pong* objects have a *content* field in which the pinging agent can place an arbitrary string. To create a valid answer pong, the pong-returning agent should copy this string into the equivalent field of the *pong* object that it returns.

```
// example Ping instantiation:
Ping p = new Ping( Secret12345 );

// example Pong instantiation, upon receipt of the above:
// assume the receive ping is in <pi>
Pong po = new Pong(pi.getContent());
```

Object

The `Communications` class permits the exchange of any Java object between agents. However, the only objects an agent is required to recognize and process are the *market message* and *atom*. It is not clear whether the Communications class should restrict the types of objects that may be communicated. Restrictions on transmitted data types would contribute to the integrity of the virtual market by limiting incomprehensible communications. However the freedom to transmit any object could allow agents to exchange data types of their own invention.

Constructor and public methods for communications

```
Communications(
        String username, String password, String identity, String popServer,
        String smtpServer,
        int pollDelayMsec, int pingHandling, int publicKeyHandling, int maxQueueSize,
        Signatures sx,
        int pongHandling, Vector _pongs, int atomHandling,
        Hashtable tradingPartners, int shortCircuitHandling,
        int defaultCounter, int defaultPointer, int defaultTimestampDelta,
        int defaultCirculationMode,
        Hashtable atomsIAnswer,
        int noChangeLimit, int hopCountLimit)
```

The `Communications` constructor sets all communications parameters such as background handling of pongs and keys. It also provides pointers to the agent-created data structures that may be maintained for the agent by the `Communications` background methods.

`void sendMessage(toID,fromID,o);`
Sends object o to agent <toID> with return address <fromID>

`void sendMessage(toID,o);`
Sends object <o> to agent <toID> with my identity as the return address

`Object retrieveMessage()`
Retrieves the next incoming message for this agent from the message queue.

`Object retrieveMessage(boolean block)`
Retrieves the next incoming message for this agent from the message queue. If <block> is true, the method blocks until a message arrives in the queue.

`String GetLastFrom()`
Returns the identity of the last message-sender

`String getMyReturnAddress()`
Convenience method returns this agent's return address, also called "identity"

`int messageCount()`
Returns the number of messages waiting in this agent's local queue

`void start ()`
Start background communications

`broadcast(MarketMessage)`
This method operates on market messages only. It sends the specified market message to every agent whose identity appears on an atom in the message. Broadcasts are flagged as such. A community-wide rule is that a broadcast message must not be rebroadcast (to avoid floods). Broadcasts are used for mass notification of all traders that a transaction candidate has been formed.

Features of the Communications class

Short circuit delivery

The only way a stateless agent can "keep alive" a message that it can't immediately process is to send the message back to itself. Short circuit delivery in that case bypasses the usual encode-and-send process and drops the message directly back into the agent's incoming queue. This avoids the overhead and delays of encoding, decoding and transmission. A brief delay (`pollDelayMsec`) was necessary to prevent short circuit deliveries from overwhelming other incoming messages.

Transport

Messages in the current version of the Atomic Market travel by conventional e-mail (SMTP and POP). Every agent has an e-mail address, password and mailbox.

E-mail is perhaps not an intuitively obvious choice, but it offers several features that are appropriate for an agent messaging system:

Globally unique address

Store and forward with queuing, automatic retry, return, forwarding

Location and address independence

Content-neutrality

Ubiquitous servers

The use of existing e-mail servers meant that no new servers had to be installed or configured for this project.

When a message is to be sent, it is serialized. The resulting `objectOutputStream` is converted to a MIME/MULTIPART e-mail extension and mailed via Sun's JavaMail API using the SMTP server that was configured when the Communications object was instantiated.

If a system such as this were deployed on a very large scale, we would likely employ servers that look very much like e-mail servers, using addresses that look very much like e-mail addresses. However, a few changes would markedly improve efficiency:

- Binary protocol, to remove expensive conversions to and from MIME
- Interrupt feature, to avoid continuous polling of mailboxes
- Direct communications, to bypass the queue when agents are online

Overall, the ideal messaging model for agent communications is roughly as shown in Fig. 7-11. All the functions are realizable through an e-mail transport or through the more sophisticated alternative proposed above.

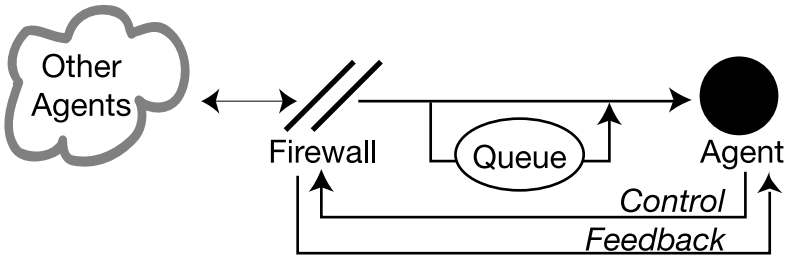


Figure 7-11: Ideal agent messaging model

The *firewall* blocks unwanted traffic, at least by the identity/address of the sender. It may perform other functions similar to those of ordinary packet-handling firewalls and routers. These additional functions may include routing messages to any of several agents on the inside, and address translation to non-routable names.

It is possible for agents to "spam" one another, though most agents will reject mail they did not expect.

Generic Agent

A configurable, general-purpose agent called the *generic agent* was designed to simplify testing and demonstration, and to facilitate the rapid construction of special purpose agents. Initial plans called for Atomic Market agents to be individually coded. Nothing in the Atomic Market architecture precludes custom-coded agents. However, early testing showed that many agent processes follow nearly identical paths, no matter what private strategies or goals are employed. Thus, the `AEXCore` was devised as a consolidator of common agent primitives, and the *generic agent* was created as a demonstration of agent programming using those primitives. The generic agent has one main goal: to receive market messages from others, alter them in compliance with its strategies, and discard them or pass them to other agents for further resolution. The generic agent is mostly stateless.

The process of encoding a general message-handling strategy in the agents involved iterative testing that revealed borderline and ambiguous cases. Once a globally useful main process loop was defined and tested, customization of each agent's capabilities through private rule sets was straightforward. The agents' behaviors are easily understood when they can be compared through examination of the rule files that conform to a standard grammar. Examination of custom source code for each agent would make examination more difficult.

Agents in an Atomic Market need not be built around the generic agent. They need only handle a large enough subset of the communication objects to be able to conduct productive pairwise interactions with other agents. However, the generic agent is a useful and instructive outline for the design of Atomic Market traders.

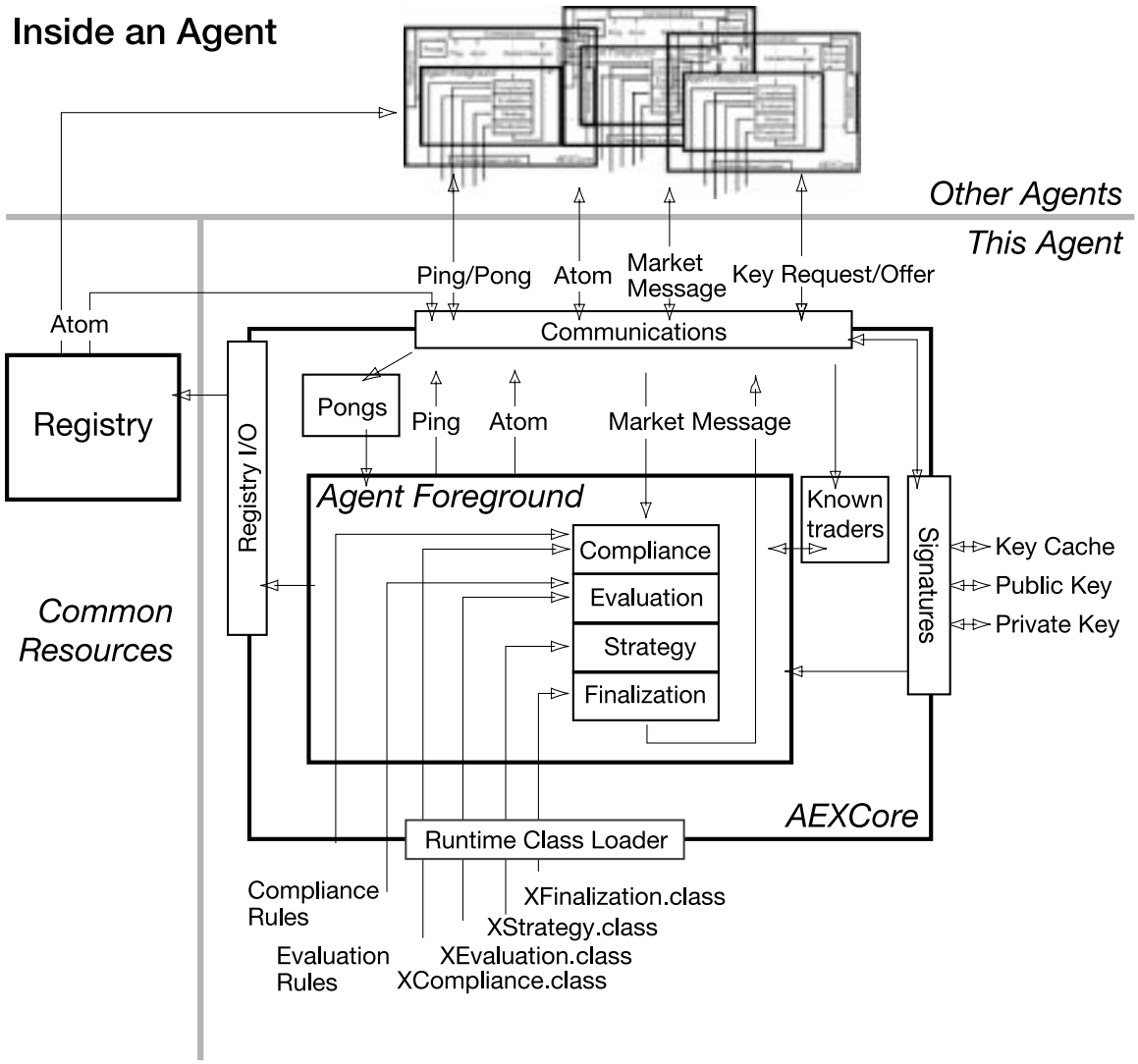


Figure 7-12: Top-down view of an agent

Types of agents

The two major classes of agents appear in an Atomic Market are: *proactive agents* and *reactive* agents. An agent may assume either or both of these roles. A single agent-running entity (a person or corporation) will probably engage several agents of both types to address a variety of tasks.

Proactive agents initiate transactions to satisfy personal goals. A personal book-buying agent is a proactive agent. A proactive agent can be prompted to action by sending it a market message that contains at least one unsatisfied term. For example, sending a market message with a lone "buy/book" atom will cause a book-buying agent to begin searching for the requested book. A *catalyst* is an external controller that generates the initial market message, called a *stimulus* that starts the agent's search process. Of course the catalyst must have access to the agent's private key in order to produce a valid market message.

Reactive agents satisfy the missing parts of market messages created by others. A retail bookseller's agent is an example of a reactive agent. In most cases there is little to be gained by telling a retail bookseller to "sell a book" so there is no action to catalyze. A book-selling agent *could* have a proactive role. For example, a seller of rare books might seek out buyers for unique items. But ordinary book-selling agents would probably assume reactive roles, addressing unfilled book orders initiated by others.

Most *proactive* agents have *reactive* behaviors as well. For example, the book-buying agent is primarily *proactive* – it seeks books to fill its own needs. However, when the message returns to the book buyer with modifications made by others, it uses reactive rules to offer money in response to a request that it pay for the books.

Proactive and reactive agents are distinguished by their behaviors, which emerge from the rules that govern them. Proactive and reactive agents have both been built using the standard generic agent, simply through adaptation of the rule files.

Proactive Agent

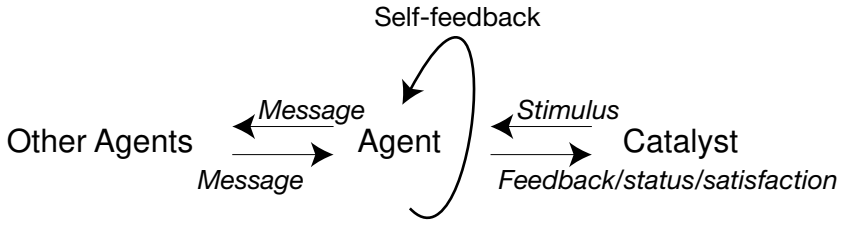


Figure 7-13: Message paths in a proactive agent

• Initiates market message negotiation via a self-owned, artificial stimulus (the *catalyst*).

• The catalyst embodies an agent-owner's desires and highest-level strategies. The catalyst may monitor the success or failure of attempted transactions, and may implement resolution strategies such as repeated attempts using the same root market message, or retries with modified initial conditions.

• Advertises in the registry to seek compatible traders for some ontology that it needs to complete (e.g. "I want a book").

• Requires protection against false external stimuli. Otherwise any agent could stimulate a book-buying agent to buy books just by sending it a false message bearing its name and the book request.

• Examples of proactive agents:

Book buyer

Stock mutual fund builder

Reactive Agent

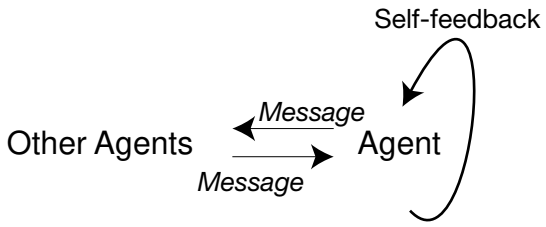


Figure 7-14: Message paths in a reactive agent

• Reacts to all market messages that come to it

• Has no catalyst: the stimulus for action is the arrival of a market message

• Connects to the registry as an event-notification customer; typically maintains a persistent presence, waiting for notice of new trading partners.

• Recognizes interesting transactions because it has rules for filling their incomplete components.

• Examples:

Bookseller, in interaction with Book Buyer
Shipper, in interaction with Bookseller
Insurer, in interaction with Shippers
Stock-proxy, in interaction with Basket Builder

Generic agent states

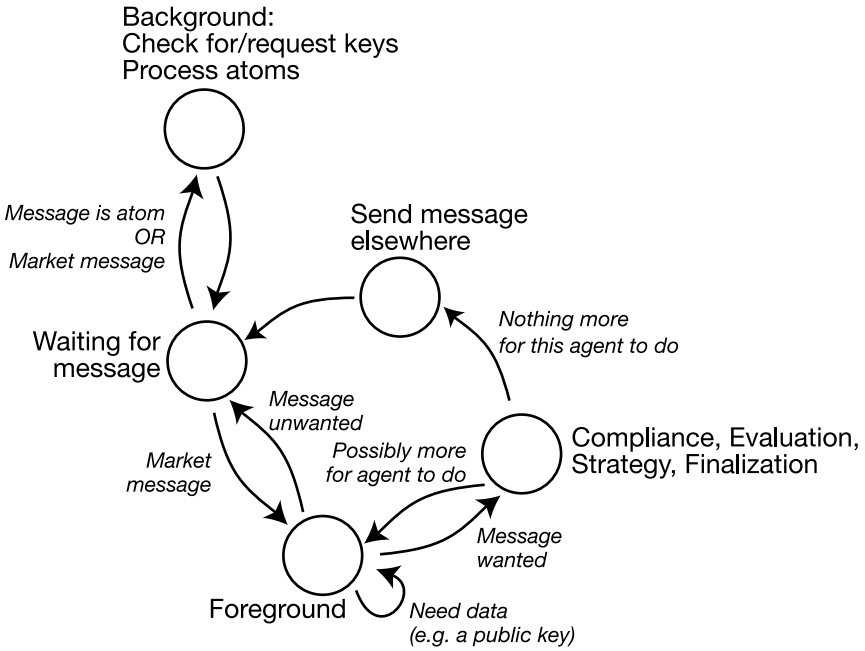


Figure 7-15: Message-handling states of a generic agent

A generic agent's state is controlled by the arrival and content of messages. Both *market messages* and single *atoms* may be received.

A *market message* is examined by both the foreground and background processes. The background process checks for the availability of public keys for every agent on the message, while the foreground process evaluates and manipulates the message based on its content. When an *atom* is received, it is processed entirely by the background process – to file a key, make note of a new trading partner, answer a ping, etc.

More detail about the foreground market message-handling process, and the routing algorithm that determines where a message will be sent next, are covered in more detail later in this chapter.

Generic agent startup

At startup, a generic agent clears any past state that might interfere with operations, and performs some registry housekeeping over its records. If it is a reactive agent, it then asks the registry to send any records of compatible traders presently seeking the services if provides.

Startup process

1. Remove placeholder keys from the local public key cache. Placeholder keys represent keys requested and not received. The agent will send a new key request if it needs the subject keys again.

2. Remove any of this agent's pending requests from the registry.

3. If this agent is a provider for some ontology, post a persistent offer to the registry as well as a persistent query request so that the agent receives a notice of any new traders who need what it offers.

4. Query the registry for any compatible traders who are already present

Catalyst: kick-start for proactive agents

The message-manipulating generic agent described here does not spontaneously generate new market messages. By design, it cannot instantiate a new market message. Problematically, the only event that can prod a proactive agent to action is the receipt of an incomplete market message. Upon receipt, a proactive agent begins the process of message manipulation, market advertising, and communication with peers to *resolve* or *complete* the message.

The generation of the incomplete message (representing just the agent's wants and usually not proposing any solutions) was moved out of the agent proper, and into a standalone companion application called the *catalyst*. Primarily, this change facilitated the use of a single *generic agent* program for all agents, whether proactive or reactive.

Why not just bind the catalyst / stimulus directly to the agent application to avoid this problem?

Stimuli might reach an agent through *many* disparate interfaces. The agent may remain in one place to execute, while its owner moves from cell phone to laptop to Internet café web browser. We would prefer to be able to contact long-running, freestanding, persistent agents by any convenient means, even (and especially) when the agent owner does not have a direct connection to the agent application.

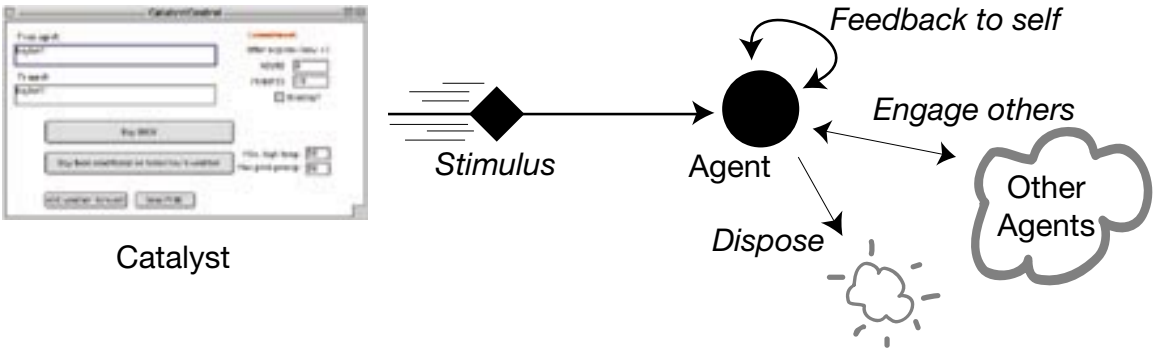


Figure 7-16: Catalyst and agent message handling options

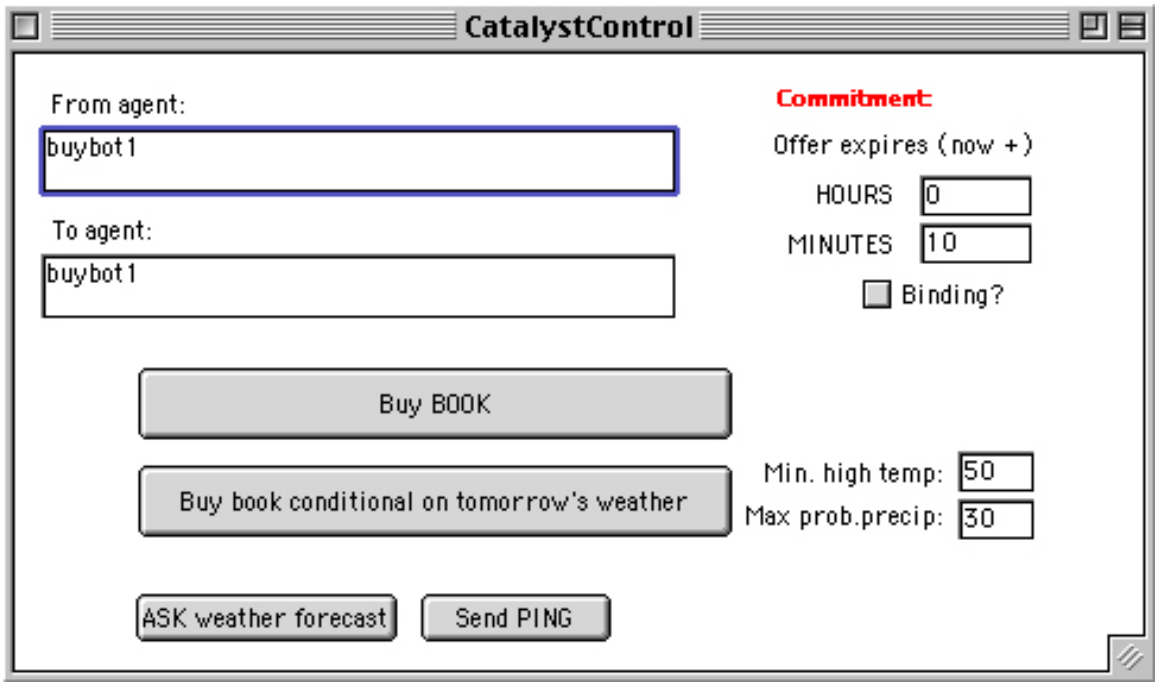


Figure 7-17: Screen shot of the Catalyst for the *buybot1* agent

Further, if certain needs were hard coded into a user's agent, then the utility of that agent as a long-running process, sensitive to external inputs, would be diminished. An agent that is able to self-start would not be well positioned to adapt to future needs not present at the time of instantiation. As built, the generic agent's rules and extended code modules can be replaced while the agent is running, creating a very versatile platform.

The Generic agent rule system

The `Rulesystem` is a package of rule-following tools that any agent may use to manipulate market messages. The current `Rulesystem` class also folds in *IBM CommonRules* for Courteous Logic Programs (CLP)[25], although CLP reasoning is not yet in the generic agent process.

The generic agent defines five steps in the resolution of a market message. Pre-written rule sets and extensible code are available for each phase:

1. Common Terms Assessment
2. Compliance
3. Evaluation
4. Strategy
5. Finalization

In the current Atomic Market, the `Rulesystem` class offers simple capabilities. Rules test preconditions only in the current component, not across the entire market message. If CommonRules were integrated into the system, it would be possible to invoke more powerful rules that react to the state of the entire market message.

Message resolution stage 1: Common terms assessment

At this stage, the agent considers whether the common terms on a market message are (1) *comprehensible* and (2) *acceptable* based on the agent's rules for common terms. Common terms can include any atom from any ontology. Because common terms bind all agents and are non-negotiable, they should include only reasonable universal constraints, such as an expiration timestamp. Any agent may add common terms to a message. However, too many or too-demanding common terms could render a message unattractive to other traders.

Message resolution stage 2: Compliance

The *compliance* stage *fills out* the transaction draft, changing its form to suit the agent. Compliance activities cover basic knowledge an agent may have about the requirements for a transaction, such as a book-selling agent's assurance that any request for a book is answered with both a book offer and a request for money. The title of the book, amount of money, and other details are handled later.

Sample rules[34]

Book buyer

```
1. ask,money,SATISFY,offer,money,false
```

Book seller

```
1. ask,book,SATISFY,offer,book,false
2. offer,money,SATISFY,ask,money,false
3. offer,shipping,SATISFY,ask,shipping,false
4. ; if there are still any loose ends, new components will be added
5. ask,book,AND,ask,money,true
6. ask,book,AND,ask,shipping,true
```

Book buyer's rule 1 is read as follows:

> "If the Market Message presents an unsatisfied ask for money, complete (satisfy) the component containing the ask with an offer of money. False means "If more than one component asks for money, fill them all."

The bookseller's rules are similarly read. Bookseller's rules 1 through 3 fill existing holes in the market message. Rules 5 and 6 can create an AND clause to bind new terms to the market message via new conjunction nodes.

[34] Line numbers do not appear in the actual rule files.

Compliance rules in action.

In the figures below, the book buyer has placed its need for a book in a new market message. The bookseller agent then responds to the need for a book, having first added its own requirements for money and shipping.

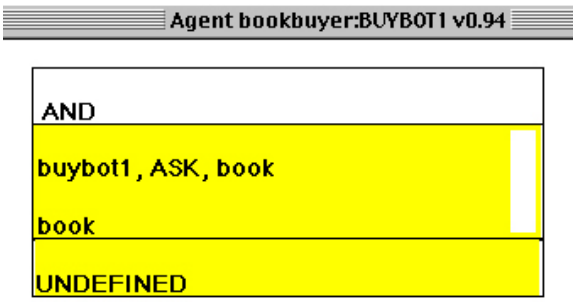


Figure 7-18: Book buyer initiates a transaction draft

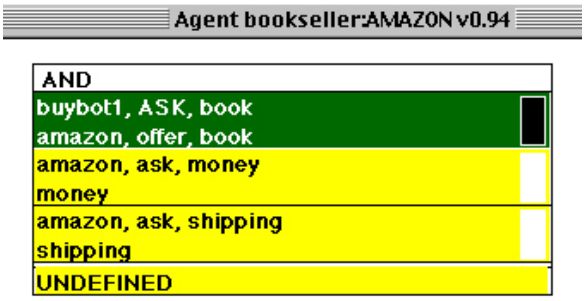


Figure 7-19: Bookseller agent adds its requirements

Message resolution stage 3: Evaluation

The *evaluation* phase examines the market message after *compliance* has manipulated it. Evaluation attempts to provide the content to fill open slots. For example, if compliance added an "offer book" component to the transaction, evaluation should supply the title of the book, its ISBN, author and so on. If a request for money was added, the price of the book should be filled during evaluation.

Examples from the bookseller agent's evaluation rules:

1. The number of books offered (%quantity) is made equal to the number requested.
"= *" copies a value from the complementary atom to the current atom:

```
%action=offer,%ontology=book:%quantity=*
```

2. The offered title is made to match the requested title:

```
%action=offer,%ontology=book:title=*
```

3. If the book is *The Lorax*, a price variable &price is loaded with the value 14.95:

```
%action=offer,%ontology=book,title=The Lorax:&price=14.95
```

4. The value stored in &price is used to fill a money atom:

```
%action=ask,%ontology=money:%quantity=&price
```

Obviously a real bookseller would not hard code book prices into a rule set, but would implement them in code via an extension to the generic agent, to provide database connectivity and other useful reasoning.

Sample evaluation rules

Book buyer

```
%action=offer,%ontology=money:%description=*
%action=offer,%ontology=money:%importance=*
%action=offer,%ontology=money:currency=*
%action=offer,%ontology=money:type=*
%action=offer,%ontology=money:%quantity=*
%action=offer,%ontology=money:%importance=0.2
```

Book seller

```
%action=offer,%ontology=book:%quantity=*
%action=offer,%ontology=book:%importance=1.0
%action=offer,%ontology=book:title=*
%action=offer,%ontology=book:isbn=*
%action=offer,%ontology=book:upc=*
%action=offer,%ontology=book:author=*
%action=offer,%ontology=book:yearpublished=*
%action=offer,%ontology=book:publisher=*
%action=offer,%ontology=book:pages=*
%action=offer,%ontology=book:editor=*
%action=offer,%ontology=book:condition=*
%action=offer,%ontology=book:genre=*

%action=offer,%ontology=book,title=The Lorax:&price=14.95

%action=ask,%ontology=money:%quantity=&price
%action=ask,%ontology=money:%importance=1.0
```

Message resolution stage 4: Strategy

The *strategy* phase provides an opportunity for reflection, assessment and revision of the market message after compliance and evaluation have run. It is a high-level assessment of the entire message and its suitability to this agent's motivations. "Does the proposed transaction suit my goals?" Strategy and evaluation may be tightly bound and may explore several alternatives before settling on an optimal result.

Strategy is implicit in the generic agent's handling of incomplete messages. Two configuration files provide, respectively, a list of atoms to which the agent can respond, and a list of ontologies for which the agent will seek other traders if it cannot answer them itself. Thus, a book-selling agent is able to seek *shipping agents* to satisfy the book-seller's request for *shipping* on a transaction draft. The rules below show that a book-seller will seek both shippers and shipping insurance, if it receives a market message that requires either, and will respond to market messages that need an offer of ontology *book*.

Bookseller, ontologies I seek	Bookseller, atoms I answer
`Shipping` `shipping_insurance`	`book, offer`
Bookbuyer, ontologies I seek	**Bookbuyer, atoms I answer**
`Book` `weather_forecast`	`(none)`

Message resolution stage 5: Finalization

In the *finalization* phase, false terms that don't impede completion of the message may be removed, and XORs are pruned so that only one term remains. Finalization may convert inequalities (<, >, <=, >=) in atoms to fixed values bound by "==". However this is not necessarily appropriate for all cases, and further investigation and experience will reveal more about the proper handling of inequalities in transaction candidates, and about the execution of transactions that do not contain definite values. Clearly, most order execution systems today expect definite values, but Atomic Market transactions may simply demand more flexibility in execution, to correspond to the flexibility in transaction design.

Market message routing

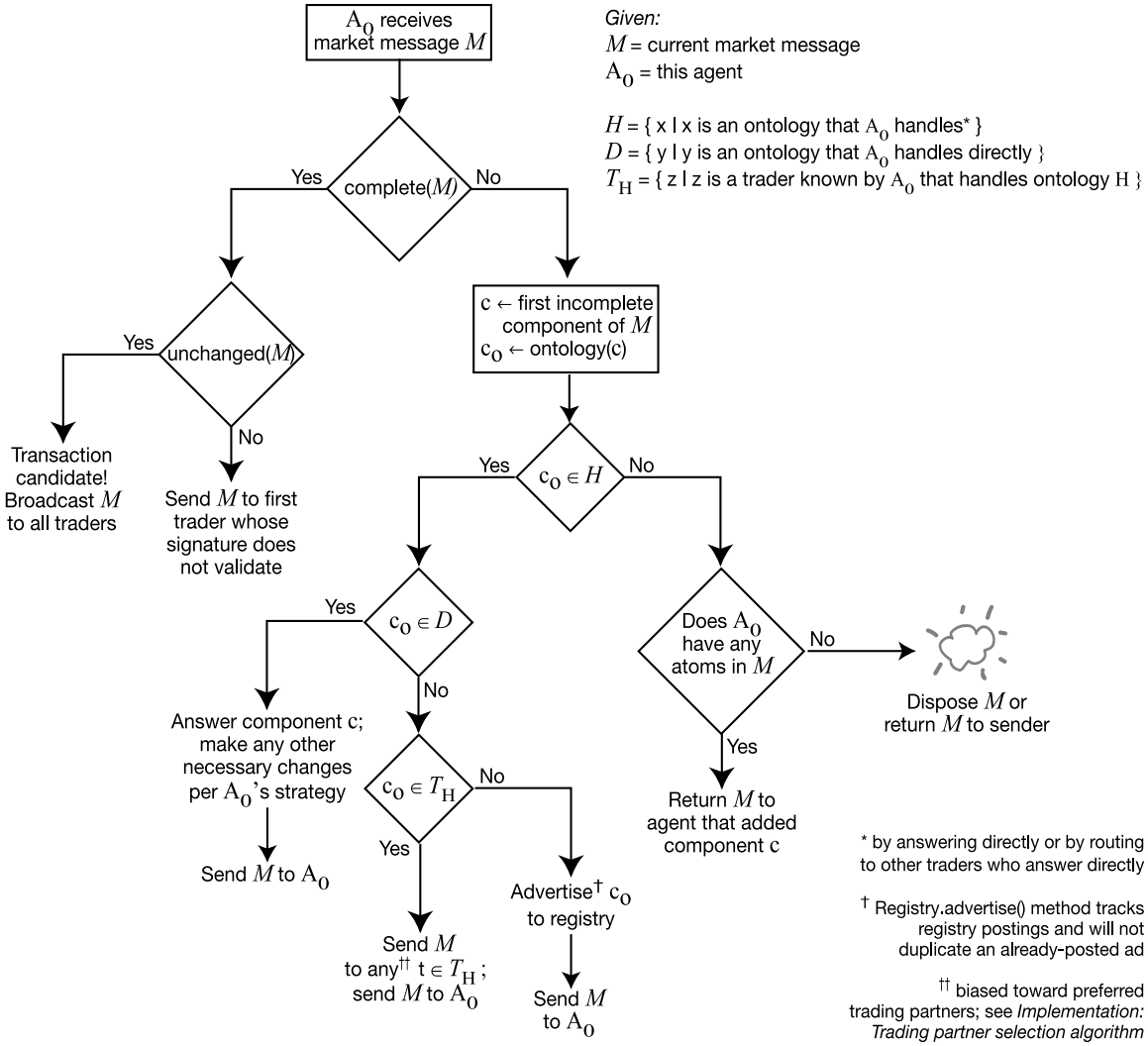


Figure 7-20: Market message routing in the generic agent

One of the most important decisions an agent must make is the determination of which trader should next receive a market message that it holds. Fig. 7-20 shows the generic agent's market message handling and routing algorithm. The agent maintains minimal state, so that it may consider only the *message*, its own private strategies, and its preference-ranked list of *known traders* to determine the next destination for a message. A trader may also examine the two *message state* records, `hopCount` and `noChangeCount`, and may discard a message if either exceeds the agent's limits. State-based limit checks are performed immediately upon receipt of a message, and are not shown in Fig. 7-20.

Dynamic extension architecture

The generic agent's rule-based subsystems are adequate for simple agents whose operations can be described by static rule files. Designers who wish to create agents that are more complex may override the standard rules with either a better rule system or procedural code. A dynamic class-loader provided in the AEXCore facilitates complex agents that use the generic agent as a root. The generic agent will load and execute external classes if the class files are appropriately located and named.

Extension classes begin with the letter "X" followed by the name of the phase they are augmenting, to wit: `XcommonTerms.class`, `XCompliance.class`, `XEvaluation.class`, `XStrategy.class`, and `XFinalization.class`. If any of these classes is present in the agent's home directory, it will be called just after the agent runs its internal rules for the stated phase (if any). For example, if a class named `XStrategy.class` is located in the agent's home directory, the class will be loaded and its `main()` method executed immediately after the agent's own internal Strategy step is run.

Extension classes currently can only alter a message. An extension module cannot discard a market message or dictate which trader should receive it next. This issue should be addressed in a future revision of the extension architecture.

A sample, skeleton XStrategy source file appears in Appendix E.

Agent trading partner selection

When an agent encounters a market message that contains an incomplete component, the agent must send copies of the message to traders that are likely to provide assistance. To facilitate its own satisfaction, the generic agent keeps a preference-ranked list of possible trading partners for the ontologies that interest it. When there is limited time to pursue trading partners, those with the highest expected probability of success should be contacted first. However, discovery of new trading partners is also important. Without ongoing discovery, and agent never learns of better traders, and the problems of inadequate search may enter the market. The generic agent undertakes a paced, incremental exploration of potential trading partners. The agent always contacts several *known* trading partners, as well as some *unknowns* that it meets through registry advertisements. The selection algorithm reflects preference-ordering with a bias toward highest-ranked traders. The algorithm is time- and result-dependent, so that as the time remaining decreases, or as the number of traders contacted approaches a limit, the probability of a low-ranked trader's selection increases to parity with the highest-ranked traders.

Trading partner selection algorithm

given h = ontology of a component under examination
T_h = {x | x is a trader known to this agent for ontology h}
n = cardinality of T_h
T_h is preference-ranked by the agent such that $x_n \in T_h$ is preferred to $x_{n+1} \in T_h$
B = bitmapped set of n bits; bit m is set if trader x_m is already part of this negotiation

ratio = Max[(time_elapsed/time_remaining),(traders_contacted/trader_limit)]
selection_size = ratio * n

p = Random selection from (1..selection_size)
if B_p is not set, set bit and contact trader B_p
if B_p is set, p = highest-ranked trader for which B_p is not set, if any

The set of possible trading partners T_h is expanded automatically by the `Communications` class when it receives solicitations from other agents. When a negotiation with a trader leads to the formation of a *transaction candidate*, all traders involved in that market message are nudged up, increasing their chances in the next negotiation. Successful traders that move up the scale climb past others that are displaced toward the bottom.

Protection strategy

In consideration of the risks to a software agent as discussed in Chapter 5, one of the first activities any agent should undertake upon receipt of a message is to validation of the message, as in this pseudo code:

```
/* retrieve a msg */
m ← getMessage()
/* if tampered, discard */
if (! m.myComponentsUnchanged(myIdentity))
then (discard message and wait for another)
```

`boolean MarketMessage.myComponentsUnchanged()` checks the signatures on all atoms bearing the agent ID `<myIdentity>`, returning `<false>` if any signatures are invalid, or `<true>` if (1) all signatures on atoms belonging to `<myIdentity>` authenticate or (2) there are no signatures on this message from`<myIdentity>`. Through this conservative rule, any message that has been tampered with is discarded without further processing.

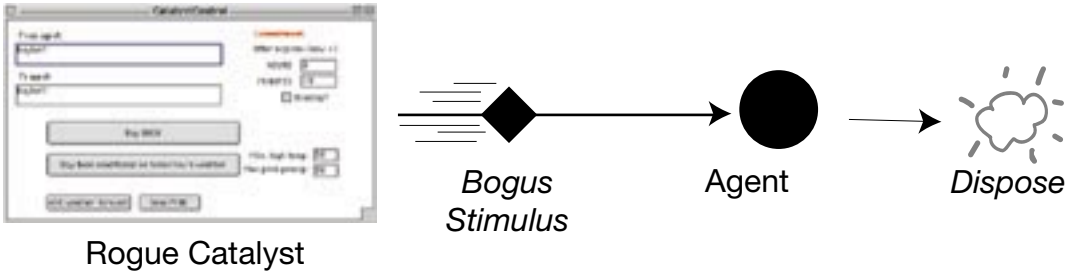


Figure 7-21: An agent rejects a bogus stimulus

An agent's quest for message completion begins with the transmission of an incomplete message from the catalyst device to the agent. We presume that legal agent-catalyzing devices have access to the agent's private key and can therefore sign the agent's messages.

Registry

The registry is the central public listing service for agents' advertisements of unmet needs and offers. The Atomic Market registry runs on a MySQL [94] client-server database, accessed by the agents through direct calls via the Java Data Access API (JDBC) [95].

The registry is not very different from other directory services in its basic operations. All registry communications are abstracted through generic methods not related to the JDBC or any other underlying interface. Agents don't interact directly with the Registry, but operate through the methods of the `Registry` class. This class provides "advertise" and "find traders" abstractions plus other methods detailed on the following pages to allow agents to maintain their listings and trigger queries. The registry class itself interacts with the database server through the `RegistryIO` class, which provides a logical interface to the true registry – the MySQL database.

Internally, the registry maintains two tables:

> *Notification*: a table of requests-for-notification. For example, a book-selling agent will ask to be notified whenever a book-buying agent appears
>
> *Advertisements*: a table of all agents looking for compatible trading partners. When an agent posts an advertisement, the posting appears in this table, and all potentially compatible traders in the notification table are notified. Any agent can also ask the registry to send it a complete set of all compatible, advertising traders by using the `Registry.FindCompatibleTraders(Atom a)` method.

Notifications from the registry to agents look exactly like direct notifications from the agents themselves. The registry uses the masquerade feature of `Communications.sendMessage` to make notifications to agents appear to have originated at the trading partner, not at the registry. Thus, the Atomic Market registry is largely invisible to agents. This approach also simplifies agents, because they do not need a separate registry listener for notifications from the registry.

Constructor and public methods of the registry

When a registry object is created, it opens a connection to the registry and maintains the connection until close() is called.

Constructor

```
Registry(     String identity, String registryAddress,
       String registryDriver, String registryMode, String smtpServer)
```

`identity`	– the agent's identity in the Atomic Market
`registryAddress`	– the address of the registry
`registryDriver`	– the driver needed to access the registry[35]
`registryMode`	– the registry can run in a "local mode" (for testing)[36]
`smtpServer`	– the server that handles the registry's outgoing messages[37]

Shutdown methods

`void close()`
closes the agent's registry connection

`void shutdown()`
shuts down a local database server [38]

Listing Methods

To post notices or query the registry through its listing and listener methods, create an atom that contains the *ontology* of the thing you seek or provide, the action you wish to perform (*ask* or *offer*) and your identity. Then call the appropriate method:

`void list(Atom a)`
advertise the Atom <a> in the registry

`void delist(Atom a)`
remove a registry listing created with list()

`void advertise (Atom a)`
similar to list(Atom a); tracks postings; will not list the same request twice

`void deadvertise(Atom a)`
remove a registry listing created with advertise()

---

[35] This is the only database-specific information the agent must have.
[36] Deprecated.
[37] Agents presently communicate via e-mail, so this is an SMTP server. However, the name is badly chosen as a non-SMTP mode is possible in future versions.
[38] Not for use with client-server databases.

Listener Methods
Instructs the registry to notify you of the arrival of compatible trader in the registry.

`void registerListener(Atom a)`
notify me whenever a compatible trader for <a> appears in the registry

`void removeListener(Atom a)`
end notifications for <a>

Query method

Instructs the registry to immediately send a one-time notification of every compatible trader for <a> that's presently listed.

`void findCompatibleTraders(Atom a)`

Monitor

The *monitor* application receives and consolidates status reports from all participating agents. The generic agent has a configuration file parameter that can optionally list the address of a monitor-agent. If an address is provided, the generic agent will send copies of all its status messages to the monitor-agent, which aggregates status messages from many agents, and displays them

```
ubermonitoraddress,aex-monitor@media.mit.edu
```

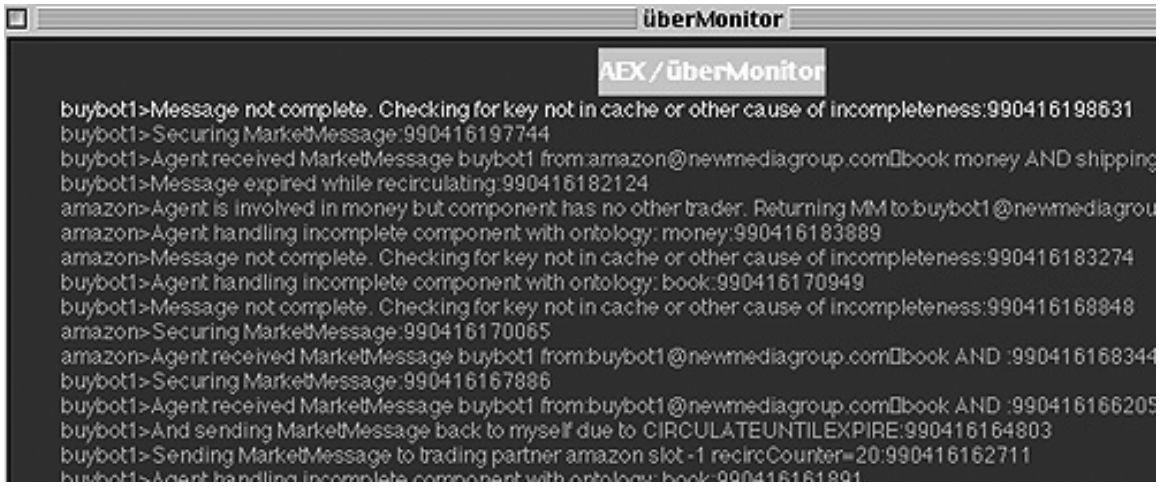


Figure 7-22: Detail of Monitor screen shot

How to run the Atomic Market

The Atomic Market core, its agents and registry are collectively referred to as the AEXCode bundle. All AEXCode is written in Java, and is compliant with Java 1.1.8 and later. It requires the Java 1.2 Swing and Collections classes that are available either directly with Java 1.2 or as Sun-provided add-in classes for version 1.1 and later of the Java Virtual Machine (JVM).

The Atomic Market was developed on Macintosh. It has been tested with the version 1.1.8JVM for Macintosh with Swing and Collections, and version 1.3 JVM on Linux /86 and Solaris/Sparc platforms.

If running AEXCode on a Java 1.2 system, it may be necessary to remove references to the pre-1.2 Swing and Collections packages and recompile, or to provide the pre-1.2 Swing and Collections classes on the target machine. Future releases of AEXCode may either incorporate the necessary classes in the AECore jar files or offer a more direct workaround.

The Swing and Collections classes for Java 1.1 are available from:

```
http://java.sun.com/products/javabeans/infobus/collectionsreadme.html
and
http://java.sun.com/products/jfc/download.html
```

Using AEXCode

1. Assemble everything needed to run an agent by installing these files in one directory:

`Agent.jar`	– the generic agent
`aexcore.jar`	– AEX code operations, nearly everything an Agent needs
`Rulesystems.jar`	– rule interpreters for the five-stage resolution process
`agent-config`	– generic agent configuration file
`/ontologies`	– a folder of common ontologies
`/strategies`	– a directory containing the generic agent's strategy files
`collections.jar`	– due to compatibility with the pre-1.2 JVM
`swingall.jar`	– due to compatibility with the pre-1.2 JVM

2. Configure and customize the agent-config file

3. Customize strategies in /strategies

4. Launch the agent:

```
java –classpath ./Agent.jar:./aexcore.jar:./collections.jar Agent
```

Ontology design

Ontologies in the Atomic Market may be one of three types: *conventional, external reference or reflective.* In addition, a fixed *root ontology* defines fields that are a part of every atom.

Root ontology

All atoms in the Atomic Market share this common root ontology. These fields are a part of every Atomic Market atom:

%ontology,string,	the name of this ontology
%description,string,	describes the contents of this atom - a label for people
%quantity,double,	how many? (dollar, items, etc)
%action,string,	legal actions are OFFER and ASK
%importance,double,	relative importance of item (i.e. probability of success if item is provided as asked)
%identity,string,	this atom's creator's identity

Conventional ontology

A conventional ontology describes an article of trade, such as a *book* or *money*. These objects are the main matter of Atomic Market negotiations. Note that the book ontology is well factored to include only fields that are strictly descriptive of a *book*. There is no field for *price* because the *money* ontology is used to construct an atom that describes an exchange of money. And the *money* ontology does not itself define a field for *quantity of money* – this is handled by the *%quantity* field inherited from the root ontology.

Book	Money
title,string	currency,string
isbn,string	type,string
upc,string	
author,string	
yearpublished,string	
publisher,string	
pages,integer	
editor,string	
condition,string	
genre,string	

External reference ontology

An external reference ontology describes external data sources that influence the outcome of transactions. The information from these sources is not likely to be negotiable, but its inclusion in a transaction draft makes clear the external dependencies of the trade. Examples of external references include weather forecasts from the National Weather Service or electrical power demand figures from the California ISO.

Weather_Forecast

city, string,	the city this forecast is for
state, string,	the state this forecast is for
date, string,	the date of the forecast
conditions, string,	Cloudy/clear/rain/fair/partly cloudy
high_temperature,	double, high temp for the day
low_temperature,	double, low temp for the day
precipitation_chance,	double, chance of precipitation

Internal reference ontology

An internal reference, or *reflective* ontology, describes rules that are contingent on the state or content of the transaction itself. Reflective ontologies can describe complex terms that don't fit into conventional atom structures, or can be used to invoke external terms that are partially dependent upon internal conditions. For example, a reflective ontology might ask a certification agent whether an insurance-selling agent is really licensed to sell insurance, or retrieve a participant's ranking from a reputation scoring service. Reflective ontologies could also test for the presence or absence of various components, thereby allowing the transaction to describe not only the items of exchange but also permissible forms for the exchange.

Reflective ontologies are not a part of the presently-implemented market. However, the design notes about the form of their implementation are presented below.

Syntax examples for reflective ontologies

IF_ANY

True if any (ontology) (field) (inequality) (value_or_field)

IF_ALL

True if all (ontology) (field) (inequality) (value_or_field)

IF_NONE

True if no (ontology) (field) (inequality) (value_or_field)

SUM

Returns numeric sum of all (field)

This could be used to count participants in a trade. It could also be used, for example, to report the total amount of money spent in a trade in which an item, shipping, insurance and other costs are negotiated separately

8 Discussion, Conclusions and Future Work

Discussion

In an Atomic Market-centered view of the future of transactions, ordinary buyers' agents will test many possible solutions before completing trades, and will negotiate with numerous peers before selecting trading partners. Such aggressive exploration could drive prices to the margin, destroying profits [96]. In agent-run markets, sellers cannot use formerly useful techniques such as expanding the search space or manipulating switching costs [97] [98] [99] to guard profit margins. Rather, they must counterbalance sophisticated searches with more sophistication: customization that capitalizes on differentiation of buyers, sellers and products [80] [81].

Software agents such as those demonstrated in the Atomic Market can create and sustain a rhythm of tension and balance in new electronic markets. Possibly, decentralized markets will reach the equilibrium of traditional markets in three-phases:

First, technology will empower buyers to broadly survey the market, running near-perfect, low-cost searches. Buyers may utilize some of their newfound capacity to express and evaluate more preference-driven searches, balancing precise results against full coverage.

Second, new traders appear in the market, leading to a proliferation of options [100]. To improve their odds of success, sellers differentiate, by combining indistinct goods into unique "bundles" [101] and expressing other unique features. As diversity increases, exhaustive searches become less practical but the customer-seller-product fit improves due to the detail provided to traders. Agents search strategically rather than by brute force.

Finally, exogenous market innovations [77] – analogous to both advertising and arbitrage in traditional markets – appear in vibrant markets as intra-market services, reducing search costs and improving market efficiency. Examples of exogenous innovations in an agent marketplace include referral agents, reputation services and demand aggregators.

Finally, the cycle repeats when the established equilibrium shifts due to the changes in the opposing forces.

Why not just use price comparison 'bots'?

This project looks toward a future of rich negotiation in which many details influence the outcome. Atomic Market transactions are unlike traditional transactions in both their detail and the dynamic manner in which they are constructed. A price comparing *bot* does not explore the true needs of the traders and merely hands the buyer to a traditional, rigid transaction system once the search ends. If human Internet shoppers with only prices at their fingertips can look beyond price when making purchase decisions, perhaps automated systems that facilitate those purchases should also represent traders' true preferences.

Forces of cooperation and competition

An open market as described here relies on partial cooperation among competing peers, a difficult goal even if the proper incentives can be created. In the Atomic Market, these incentives are implicit in the workings of the system. The message-building scheme fans out a proposed transaction to many trading partners, much as a redundant network passes data along multiple paths. For every market message a trader receives, it knows some competitors have received it too. To maintain its relevance in the marketplace, a trader must handle incoming market messages carefully. It should discard messages that are misdirected or incomprehensible, or that make impossible demands. Otherwise, it should provide competitive, considered replies. Having full knowledge of its own strategy but knowing little of the strategies of its peers, the most a trader can assume about any *other* trader is that the other will adopt a similar strategy.

Intra-market traders

A distributed marketplace such as the Atomic Market would probably attract a large population of *middle agents* that act entirely within the market rather than representing buyers or sellers. These traders are brokers, recommenders, reputation systems, and monitors that improve a market by providing intermediary services. The marketplace would also attract other middle agents, such as demand aggregators like Mercata.com.

However, *Mercata the Atomic Market service provider*, unlike *Mercata the web site*, would not need to draw traders to an exclusive web marketplace. Rather, a Mercata agent would profit from through two service offerings. First, it could emulate the Mercata web site by representing itself as a seller, collecting orders from many buyers and placing a bulk order on their behalf with a real

seller. Second, it could sell aggregation services. A party planner might use a Mercata service to accumulate orders for baseball tickets, then transmit the bulk order to a sales agent.

Transactions could also involve official services. For example, a state insurance authority could certify that a particular sales agent is authorized to write automobile insurance. The direct real-time connection between every transaction and the appropriate authority would provide more protection to consumers than they currently have, because every vendor could be validated before the completion of every transaction.

Are agreements binding?

A transaction candidate could bind its traders with the legal force of a contract under the Uniform Electronic Transaction Act (UETA), an extension of existing law known as the Uniform Commercial Code (UCC) [102]. Under UETA, an agent's acts on behalf of a human or corporate principal are treated as acts of the principal itself.

To be binding, transactions and contracts must be *signed.* As interpreted by these acts, a *signature* is a participant's indication of its willingness to participate in the transaction. A signature need not be a literal handwritten signature, nor even a digital signature. The presence of components placed on the transaction by a participant may themselves constitute a signature. The Atomic Market's use of cryptographic signatures to secure the entire transaction does strengthen the evidence that a trader intentionally placed certain components on a transaction. However, it is important to be clear that the UETA concept of *signature* does not demand a digital signature. UETA explicitly disclaims any intent to be a "digital signature statute." [39]

[39] UETA §9 provides additional guidance regarding the definition of *signature*, and the act of *signing*.

Resource consumption

Software agents consume both CPU time and bandwidth. To aid understanding of the feasibility of Atomic Market-like market systems for everyday trading, measures of the approximate bytes and messages transferred, and gross CPU overhead for compute-intensive tasks, were made. The table on the next page considers a search for a particular book by three methods:

(1) A manual search at three web sites

(2) An 18-site search by a conventional price-shopping bot

(3) Projected results for an Atomic Market search with 18 booksellers that in turn contact 10 shippers

Comparison of projected Atomic Market searches to traditional methods suggests that Atomic Market agents will make reasonably efficient use of capacity, comparable to present methods, with efficiency potentially increasing as agents align with preferred trading partners.

Assumptions for the Atomic Market projection:

Agents fan out messages to approximately the same number of traders a shopping bot would contact (18)

Each agent that solicits “bids” only sends back its best offer to the agent that first contacted it (e.g. the bookseller agent uses only the “best” shipping quote in its response to the book buyer)

The book buyer and 50% of the bookseller agents consult the registry

Figure 8-1: Resources consumed by the search for a book

Task	*(1) Manual*	*(2) Manual w price bot*	*(3) Atomic Market typical case †*
Load Application	C0	C0	C0
Generate book request			C4
Book request to registry			4K
Registry notice to sellers			0.6K x 18
Solicitations to buyer			0.6K x 18
Market msg to sellers			4K x 18
Load shopbot home page		151K	
Load subject "book" page		116K	
Load bookstore home page	148K x 3	148K x 18	
Search for book	(C1) x 3	(C1) x 18	(C1) x 18
Shipper req. to registry			4K x 18 x 0.5
Registry notice to shippers			0.6K x 10 x 18 x 0.5
Shippers solicit sellers			0.6K x 10 x 10 x 0.5
List of everything	23K x 3		
"see all books"	91K x 3		
Retrieve search results		116K x 18	
Parse results		(C3) x 18	
Display bot results		332K	
Market msg to shippers			5K x 10 x 18
Market msgs from shippers			5K x 10 x 18
Market msgs to buyer			5K x 18
"About this book"	116K x 3	116K	
Add to shopping cart	29K	29K	
Proceed to checkout	66K	66K	
Shipping address	23K	23K	
Select shipping method	23K	23K	
Select payment method	23K	23K	
Credit card info	23K	23K	
Review order	23K	23K	
Order completion	23K	23K	
Credit card processing	(C2)	(C2)	
Bytes transferred*	1,367K	5,700K	2,137K
Messages transferred**	46	168	632
CPU time	C0+3C1+C2	C0+18C1+C2	C0+18C1+C2+C4
Incoming msgs			
… to buyer			36 to 180
… to a single seller			12
… to a single shipper			36

† Assumes warm start – that some agents know trading partners (see discussion)
* Not including page requests from browser to server
** Including page requests from browser to server

What is the information capacity for negotiations?

Atomic Market interactions have an additional, unique load in the generation and validation of signatures over both entire market messages and their components. The signature-related overhead is substantial, dominating most other processing. From the point of view of an agent, this overhead creates the incentive for the agent's peers to communicate selectively, but also reduces local capacity for strategy and exploration. Measurements were made on an Apple PowerBook G3/333 using Java 1.1.8 from the MRJ 2.2 standard installation.

Foreground processing time per message:

	msec
Evaluation and compliance	11
Securing / signature generation	107
Check other signatures	140
Complete / transmit message	4
Total time/ message, on average:	262
Messages per agent in 10 minutes:	2,290 ideal case

* *One foreground process per agent, with non-blocking background message I/O (the overhead of background message handling is not considered here)*

Comments about resource measures

Will 2,000 messages per transaction take an agent through enough of the market to assure good prices and access to goods *provided* other agents ahead of it do not feed back excessively? This figure seems reasonable considering that the book buying agent in the previous table would have handled between 36 (best case) and 180 (unlikely worst case) messages for a two-level search that touched (18 + 10 = 28) other agents. Computational overhead is sufficiently high that communication speed is not a significant factor in capacity estimation. More sophisticated agents in the Atomic Market would have *evaluation and compliance* stages that account for a proportionally higher share of total overhead. The total overhead will increase relative to the complexity of the generated transaction drafts, and could serve as a measure of the relative sophistication of the agent's trading partners.

Conclusions

This thesis has presented the Atomic Market, one of the first implementations of a decentralized peer-to-peer agent marketplace. Theoretical and practical considerations of the system were discussed, in comparison to traditional centralized systems, as was the system's approach to resolving the self-interests of multiple agents through iterative cycles of contract revision.

A catalog of plausible attacks against the newly proposed system was presented, with suggested protective measures. Also presented was a *network model* for e-commerce, in which agent traders are viewed as network nodes, and their messages as datagrams, facilitating certain aspects of agent design and analysis.

Capacity and overhead measurements of the demonstration system suggest that despite the high message counts of Atomic Market interactions, the market's processing and communications demands are roughly analogous to the overhead of present-day e-commerce interactions.

Deployment

The Atomic Market breaks with existing practice in electronic markets. Though its flexible interaction structures are reminiscent of practice in traditional face-to-face markets, the adaptation for online trading is probably not familiar to traditional traders. Further, more work is needed to define appropriate human interfaces to agent-run market systems that are controlled by strategies and guidelines rather than literal orders.

If the Atomic Market were deployed today, how would that deployment occur? The system could be grafted onto current e-commerce systems as a front-end for agent-to-agent negotiation, and the relevant business policies and strategies identified and encoded. Presumably, such strategies would begin simply and increase in complexity as both agents and their owners gained experience. Perhaps organizations would first experiment with Atomic Market-like systems internally, using them to allocate resources or inventories. Eventually, two trading partners who are both using such a system internally might reconcile their private ontologies and link the systems. In this way, the market-as-network could expand from the inside out with the support of motivated traders.

Future Work

The examples presented in this thesis were small, easily diagrammed cases. However, the Atomic Market as currently designed will also support interactions that are much more complex. The unanswered issues for such interactions include the design of appropriate interfaces for the management, monitoring and configuration of flocks of agents rather than individuals. As well, tools are needed to rapidly generate the configuration and rule files that make every agent unique.

Registry

Messages *to* the registry (for listing and queries, as opposed to notifications from the registry) presently use a direct connection from every agent to the central MySQL database via JDBC. Though it is heavily abstracted, this is less than ideal for two reasons. First, agents must speak two protocols, the Atomic Market agent-to-agent protocol, and the registry protocol. Second, agents incorporate database access code as part of the `AEXCore`, and could accidentally damage the registry. In the future, registry access should be implemented through a peer agent just like all other messaging. A registry agent could completely hide details of registry operations, and a peer-to-peer registry structure would allow many registries and referral services to coexist transparently, potentially creating a market for referral and directory services. Advanced referral agents might offer screening and other efficiency-increasing services. In practice, agents would continue to call registry methods as they do now.

Additionally, for security reasons, the registry should authenticate signatures over atoms before accepting commands, but at present does not provide this functionality.

Signatures

The Signatures system seems to have a problem exchanging or validating signatures across platforms. It is not clear if this is a problem with the `AEXCore` methods, with multiple versions of the JVM and Java Cryptography packages, or with a flaw in the `AEXCore` packaging of the necessary components. The `AEXCore` at present does not guarantee that components will be packed up for signing in a consistent manner. The fields of atoms (and the atoms within components) should be sorted in an unambiguous, repeatable fashion before signature generation.

Generic agent

When reading the `agent-config` file, the generic agent should only require the agent's full identity ("agent@domain.com"). It currently requires both the agent's identity (its full e-mail address with user@domain) and its POP user-id, because messages are transported by e-mail. This is a technically proper approach due to the diversity of mail servers and the possibility that an e-mail address and POP user-id may not be identical. However, it exposes too much of the communications infrastructure to the agents. To facilitate the transparent conversion to other communications infrastructures in the future, agent POP user-ids and e-mail addresses should be required to be either identical or structured so that one may be derived from the other. With such a change in place, agent communications may be converted to other methods simply by changing the `Communications` class and revising the agent identity line in the `agent-config` file.

Market Message

Traders store non-negotiable atoms that bind all participants in the market message's *common terms* section. Atoms commonly found in a common terms section would carry commitment terms, expiration dates and specifications for transaction clearing. At present, the common terms are simply listed, with an implied 'and' between them. A tree-based common terms section similar to a transaction draft should be considered. Then both the terms of the transaction, and its content, could be fully negotiable. The existing methods for handling transaction drafts could be adapted to accommodate this enhancement.

Private state data should be explicitly zeroed when a message leaves an agent, to protect against leaking information about the agent to competitors. The private state data fields are by definition not intended to carry information reliably from agent to agent.

"Mark for deletion" could be the first feature of more elaborate collaborative editing tools. If a component were marked for deletion and the message were fully signed with the deletion mark intact, then the item could be deleted with the implied approval of all the traders. Thus, negotiations could propose not just alternate transaction content, but also alternate forms.

AEXCore

To determine that a message is "complete" (all conditions satisfied) an agent should only have to look at the truth value of the top-most component. Atom truth values are connected cleanly

through to components, and components are always arranged in a tree structure. The present code inefficiently steps through all the components of a market message to assess the truth value of the message.

More work is needed to understand the appropriate *segmentation* for Atomic Market communications, and to determine more precisely the role of each layer and each layer's mapping to a place on the ISO stack.

References

1. Smith, R. *The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver*. IEEE Transactions on Computers, 1980. 29(12): p. 1104-1113.
2. Sandholm, T. and Lesser, V. *Issues in Automated Negotiation and Electronic Commerce: Extending the Contract Net Framework*, First International Conference on Multiagent Systems (ICMAS95). 1995. San Francisco: AAAI Press and MIT Press. pp. 328-335.
3. Chan, E. and Swatman, P.M.C. *Electronic Commerce: A Component Model*, 2000, Department of Information Systems, Victoria University.
4. Maes, P., Guttman, R.H. and Moukas, A.G. *Agents that Buy and Sell: Transforming Commerce as We Know It*. Communications of the ACM, Mar. 1999. 42(3): p. 81.
5. Borzo, J. *LeatherXchange.com Faces Large but Wary Market*, Wall Street Journal. Jan. 5, 2001: New York.
6. Gutscher, C. *Paper Industry is Reluctant to Give Role to Independent E-Exchanges*, Wall Street Journal. Jan. 5, 2001: New York.
7. Krasner, J. *Delisted means you're dead*, Boston Globe. May 27, 2001: Boston. p. E01.
8. Fai, T.W. *Large firms cite benefits of e-commerce*, Business Times. June 19, 2000: Singapore.
9. Benson, C. *People in Business*, San Francisco Chronicle. Oct. 20, 2000. p. B5.
10. Tessler, J. *E-Business Buzz*, San Jose Mercury News. Oct. 25, 2000: San Jose.
11. The Internet Society *The Internet Open Trading Protocol - IOTP, Version 1.0*. April, 2000, The Internet Society, http://www.landfield.com/rfcs/rfc2801.html.
12. Coleman, M. *Services, Not Just Software, are Key to B2B Marketplaces*. Investor's Business Daily. Dec. 1, 2000. p. A10.
13. Stewart, A. *E-commerce 'will change the world'*. Financial Times. Oct. 18, 2000: London. p. 12.
14. Brynjolfsson, E. and Smith, M.D. *Frictionless Commerce? A Comparison of Internet and Conventional Retailers*. Aug. 1999, MIT Sloan School of Business. http://ecommerce.mit.edu/papers/friction.
15. Schmitz, S.W. *The Effects of Electronic Commerce on the Structure of Intermediation*. 2001, Austrian Academy Of Sciences Research Unit For Institutional Change And European Integration - ICE Working Paper Series. p. 36.
16. Rosenschein, J.S. and Zlotkin, G. *Rules of Encounter: designing conventions for automated negotiation among computers*. 1994, Cambridge, MA USA: MIT Press.
17. Marsella, S., Adibi, J., Al-Onaizan, Y., Kaminka, G., Muslea, I. and Tambe, M. *On being a teammate: Experiences acquired in the design of RoboCup teams*. Autonomous Agents 99. 1999. Seattle, WA USA: ACM. pp. 221-227.

18. Pynadath, D.V., Tambe, M. and Chauvat, N. *Building Dynamic Organizations of Distributed, Heterogeneous Agents*. Proceedings of Fourth International Conference on Autonomous Agents. 2000. Barcelona, Spain: ACM. pp. 195-196.

19. Hardin, G. *Tragedy of the commons*. Science, 1968. 162: pp. 1243-1248.

20. Zacharia, G., Moukas, A. and Maes, P. *Collaborative Reputation Mechanisms in Electronic Marketplaces*. Proceedings of 32nd Hawaii International Conference on System Sciences. 1999. Hawaii.

21. Parkes, D.C. and Ungar, L.H. *The Tragedy of the Commons: Pricing Social Welfare in Multiagent Systems*. 1996. unpublished report, University of Pennsylvania: Philadelphia, PA.

22. Dignum, F., Dunin-Keplicz, B. and Verbrugge, R. *Agent architecture and theory for team formation by dialogue*. April 2000, Prepublication submission to ATAL 2000. University of Groningen, Netherlands.

23. Wooldridge, M. and Jennings, N.R. *Formalizing the cooperative problem solving process*. Proceedings of Thirteenth International Workshop on Distributed Artificial Intelligence. 1994. pp. 403-417.

24. Grosof, B., Labrou, Y. and Chan, H. *Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML*. Proceedings of the first ACM Conference on E-Commerce. 1999. Denver.

25. IBM Corp. *IBM CommonRules*, http://alphaworks.ibm.com/tech/commonrules.

26. Ginis, R. and Chandy, K.M. *Micro-Option: A Method for Optimal Selection and Atomic Reservation of Distributed Resources in a Free Market Environment*. Proceedings of the 2nd ACM Conference on Electronic Commerce. Oct. 2000. Minneapolis.

27. Georgeff, M.P., Pell, B., Pollack, M.E., Tambe, M. and Wooldridge, M. *The Belief Desire-Intention Model of Agency.* 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures and Languages (ATAL-98). 1998.

28. Arrow, K.J. *Social Choice and Individual Values*. 2nd ed. 1963, New Haven: Yale University Press.

29. Shoham, Y. and Tennenholtz, M. *On the Synthesis of Useful Social Laws for Artificial Agent Societies*. Principles of Knowledge Representation and Reasoning: Third International Conference. July 1992. San Jose, CA USA.

30. Tennenholtz, M. and Moses, Y. *On Cooperation in a Multi-Entity Model*. International Joint Conference on Artificial Intelligence. 1989. Detroit, MI. pp. 918-923.

31. Walker, A. and Wooldridge, M. *Understanding the Emergence of Conventions in Multi-Agent Systems*. First International Conference on Multiagent Systems (ICMAS 95). 1995. Manchester, UK.

32. Dellarocas, C. and Klein, M. *Civil agent societies: Tools for inventing open agent-mediated electronic marketplaces*. IJCAI workshop. 1999.

33. Dellarocas, C., Klein, M. and Rodriguez-Aguilar, J.A. *An Exception-Handling Architecture for Open Electronic Marketplaces of Contract Net Software Agents*. Proceedings of the ACM Conference on Electronic Commerce (EC00). Oct. 17-20, 2000. Minneapolis.

34. Guttman, R.H., Moukas, A.G. and Maes, P. *Agent-Mediated Electronic Commerce: A Survey*. Knowledge Engineering Review, June 1998. 13(2): pp. 143-152.

35. Chavez, A. and Maes, P. *Kasbah: An Agent Marketplace for Buying and Selling Goods*. First International Conference on Practical Applications of Intelligent Agents and Multi-agent Technology. April 1996. London.

36. Guttman, R.H. and Maes, P. *Agent-mediated Integrative Negotiation for Retail Electronic Commerce*. Workshop on Agent Mediated Electronic Trading (AMET98). May 1998.

37. Guttman, R. and Maes, P. *Cooperative vs. Competitive Multi-Agent Negotiations in Retail Electronic Commerce*. Second International Workshop on Cooperative Information Agents (CIA98). July 1998. Paris.

38. Tsvetovatyy, M., Gini, M., Mobasher, B. and Wieckowski, Z. *MAGMA: An agent-based virtual market for electronic commerce*. Journal of Applied Artificial Intelligence, Special Issue on Intelligent Agents, 1997. 11(6).

39. Vulkan, N. and Jennings, N.R. *Efficient Mechanisms for the Supply of Services in Multi-Agent Environments*. First International Conference on Information and Computation Economies. 1999. Charlestown, SC., pp. 1-10. ftp://ftp.elec.qmw.ac.uk/pub/isag/distributed-ai/ publications/ice98.ps.

40. Jennings, N.R., Faratin, P., Johnson, M.J., O'Brien, P. and Wiegand, M.E. *Using intelligent agents to manage business processes*. First international conference on the practical applications of intelligent agents and multi agent technology. April 1996. London, UK.

41. Mullen, T. and Wellman, M.P. *A Simple Computational Market for Network Information Services*. First International Conference on Multiagent Systems (ICMAS-95). 1995. San Francisco, CA.

42. Wellman, M.P, Walsh, W.E., Wurman, P.R. and MacKie-Mason, J.K. *Auction Protocols for Decentralized Scheduling,* revised and extended version of *Some economics of market-based distributed scheduling*. 18th International Conference on Distributed Computing Systems. May 1998. Amsterdam.

43. Walsh, W. and Wellman, M. *Modeling Supply Chain Formation in Multiagent Systems*. IJCAI Workshop on Agent Mediated Electronic Commerce. 1999.

44. Wurman, P.R., Wellman, M.P. and Walsh, W.E. *The Michigan Internet AuctionBot: A Configurable Auction Server for Human and Software Agents*. Autonomous Agents 98. 1998. Minneapolis, MN.

45. Portillo, E. and Ahmed, P. *Design methodology for secure distributed transactions in electronic commerce*. Computer Standards & Interfaces, May 25, 1999. 21(1): pp. 5-18.

46. Ariba *B2B Marketplaces in the New Economy*. 2000, Mountain View, CA.

47. Young, E. *Falling Star*. Industry Standard Magazine. April 16, 2001.

48. Izodia http://www.izodia.com/americas/default.asp.

49. VerticalNet http://www.verticalnet.com/.

50. VerticalNet *VerticalNet defines strategy to focus on creation of private marketplaces that connect to enabled supplier networks*. Press release, May 18, 2001. http://www.verticalnet.com/news/PressReleases.asp.

51. FreeMarkets.com *FreeMarkets Conducts First-Ever Online Reverse Auctions in India*. press release, Dec. 8, 2000, FreeMarkets.

52. Gallien, J. *Smart B2B Auctions*. Short talk, MIT Sloan School of Business, Mar. 29, 2000. http://web.mit.edu/jgallien/www/research.html.

53. Frictionless Commerce *Frictionless Commerce Takes Command of Strategic Sourcing*. press release, May 12, 2001. http://www.frictionless.com/news/pr_20010205.html.

54. Osborn, S. *The role of agents in business to business (B2B) commerce*. AgentLink News, Jan. 2001. 6: pp. 6-8.

55. Jennings, N., Woghiren, K. and Osborn, S. *Interacting Agents – the way forward for Agent-Mediated Electronic Commerce*. Lost Wax. June 2000. http://www.lostwax.com/lostwax1/opinions/white-papers/int_agents/Interacting%20Agents.pdf.

56. Henderson, K. *TruExchange Debuts Trading Engine*. eMarketect – The Journal for eMarketplace builders. Dec. 5, 2000. http://www.emarketect.com/headlines/show_headlines.cfm?id=19.

57. Moss, F. *Business web automation: A winning strategy for plug-and-play e-commerce*. Nov. 2000, Bowstreet, Inc. white paper: Lynnfield, MA. p. 16.

58. Bowstreet *Bowstreet Business Web Factory: How business webs, web services and automation are transforming B2B interactions on the web*. 2000, Bowstreet, Inc. Portsmouth, NH.

59. Ricadela, A. *Bowstreet see end to hardwired apps: Software-component metadirectory lets companies quickly customize web applications*. InformationWeek. Feb. 5, 2001.

60. Ritchie, E. *eBay Opens Up*. Dec. 6, 2000, AuctionWatch.com.

61. Roth, D. *Meet Ebay's Worst Nightmare*. Fortune, June 26, 2000. 142(1): p. 199.

62. Hogan, M.D. and Radack, S.M. *The quest for information technology standards for the global information infrastructure*. StandardView, 1997. 5(1): pp. 30-35.

63. Rehesaar, H. *International standards: Practical or just theoretical?* StandardView, Sept. 1996. 4(3): p. 123-127.

64. Genesereth, M., Fikes, R. et al *Knowledge interchange format version 3.0 reference manual*. 1992. Stanford University technical report: Stanford, CA. http://logic.stanford.edu/kif/Hypertext/kif-manual.html.

65. Finin, T.F., McKay, D. and McEntire, R. *KQML as an Agent Communication Language*. CIKM '94. Nov. 1994. Gaitherburg MD.

66. Neches, R., Fikes, R., Finin, T., Gruber, R., Patil, R., Senator, T. and Swartout, W. *Enabling Technology for Knowledge Sharing*. AI Magazine, Fall 1991. 12(3): pp. 36-56.

67. Ginsberg, M. *The Knowledge Interchange Format: The KIF of Death*. AI Magazine, Fall 1991. 12(3): p. 57-63.

68. Markoff, J. *Plan Aims to Foster Electronic Commerce Between Businesses.* The New York Times. Sept. 5, 2000, late edition. p. C4.

69. Commerce.net http://www.commerce.net.

70. Tenenbaum, J.M., Chowdry, T.S. and Hughes, K. *Eco System: An Internet Commerce Architecture.* IEEE Computer. May 1997. pp. 48-55.

71. Glushko, R.J., Tenenbaum, J.M. and Meltzer, B. *An XML framework for agent-based e-commerce*. Communications of the ACM, Mar. 1999. 42(3): pp. 106-114.

72. *Security Services Markup Language, S2ML*. http://www.s2ml.org/.

73. vanHarmelen, F. and Horrocks, I. *The Ontology Inference Layer for the semantic web*. IEEE Intelligent Systems, Dec. 2000. 15(6): pp. 69-72. http://www.ontoknowledge.org/oil/.

74. Webber, D. and Dutton, A. *Understanding ebXML, UDDI and XML/EDI.* Oct. 2000, http://www.xml.org/feature_articles/2000_1107_dutton.shtml.

75. Ballinger, K. *Web Services Interoperability and SOAP*. Microsoft Developer Network Library, May 1, 2001. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsoap/html/soapinteropbkgnd.asp.

76. Wolter, R. *Overview of SOAP Client in Windows XP*. Microsoft Developer Network Library, May 2001. http://msdn.microsoft.com/library/default.asp?URL=/library/techart/xpsoap.htm.

77. Stigler, G.J. *The Economics of Information*. Journal of Political Economy, 1961. 69(3): pp. 213-225.

78. Smith, A. *Wealth of Nations*. bartleby.com, 1776. ed. C.J. Bullock. Vol. X. 1909-14, New York: The Harvard Classics; P.F. Collier & Son.

79. Wurman, P.R., Wellman, M.P. and Walsh, W.E. *The Michigan Internet AuctionBot: A configurable auction server for human and software agents*. Proceedings of the 2nd International Conference on Autonomous Agents. 1998. Minneapolis: ACM Press.

80. Chamberlin, E. *The Theory of Monopolistic Competition*. 1933. Cambridge, MA: Harvard University Press.

81. Diamond, P.A. *A Model of Price Adjustment*. Journal of Economic Theory, 1971. 3: pp. 156-168.

82. Postel, J. *DARPA Internet Program Protocol Specification*. 1981. http://www.faqs.org/rfcs/rfc791.html.

83. NASDAQ *About Nasdaq*. Nasdaq online reference accessed May 7, 2001, National Association of Securities Dealers: New York. http://www.nasdaq.com/about/about_nasdaq.stm.

84. NASDAQ *Nasdaq in Black & White 2000*. 2000, National Association of Securities Dealers: New York.

85. National Institute of Standards and Technology (NIST) *Federal Information Processing Standards Publication 180-1, Secure Hash Standard.* April 17, 1995. National Institute of Standards and Technology. http://www.itl.nist.gov/fipspubs/fip180-1.htm.

86. McCullagh, A. and Caelli, W. *Non-Repudiation in the Digital Environment*. First Monday, July 19, 2000. 5(8). http://firstmonday.org/issues/issue5_8/mccullagh/index.html.

87. McCullagh, A., Caelli, W. and Little, P. *Electronic Signatures – Understand the Past to Develop the Future*. University of New South Wales Law Journal, 1998. http://www.law.unsw.edu.au/unswlj/ecommerce/mccullagh.html.

88. Basely, W.D. *The Email Abuse FAQ*. June 25, 1998. http://members.aol.com/emailfaq/emailfaq.html.

89. Zimmermann, H. *OSI Reference Model – The ISO Model of Architecture for Open Systems Interconnection*. IEEE Transactions on Communications, 1980. 28(4): pp. 425-432.

90. Egevang, K. and Francis, P. *The IP Network Address Translator (NAT)*. May 1994, The Internet Society Network Working Group. http://www.faqs.org/rfcs/rfc1631.html.

91. Mazières, D. and Kaashoek, M. *The design, implementation and operation of an email pseudonym server*. 5th ACM conference on computer and communications security. Nov. 2-5, 1998. San Francisco, CA. pp. 27-36.

92. Schneier, B. *Applied Cryptography*. 1995: John Wiley & Sons.

93. Sun Microsystems *Java Security API.* http://java.sun.com/security/.

94. MySQL AB *MySQL.* http://www.mysql.com/.

95. Sun Microsystems *Java Data Access API.* http://java.sun.com/jdbc/.

96. Bertrand, J.L.F. *Théorie mathématique de la richesse sociale par Léon Walras: Recherches sur les principes mathématiques de la théorie des richesse par Augustin Cournot*. Journal des savants, 1883. 67: pp. 499-508.

97. Nelson, E. *While Brand Managers Embrace Variety, Too Many Choices Frustrate Consumers.* The Wall Street Journal, Apr. 20, 2001: New York.

98. Anderson, S.P. and Renault, R. *Pricing, product diversity, and search costs: a Bertrand-Chamberlin-Diamond model*. RAND Journal of Economics, 1999 (Winter). 30(4): pp. 719-735.

99. Stango, V. *Competition and Pricing in the Credit Card Market.* The Review of Economics and Statistics, Aug. 2000. 82(3): pp. 499-508.

100. *AOL, Yahoo Jump Aboard Online-Marketplace Bandwagon.* Mar. 21, 2000. San Jose Mercury News. 2000: San Jose.

101. Vulkan, N. E*conomic Implications of Agent Technology and E-Commerce.* 1998. Department of Economics, University of Bristol, 8 Woodland Road, Bristol BS8-1TN, U.K., and The Centre for Economic Learning and Social Evolution (ELSE), University College London: London. p. 36.

102. National Conference of Commissioners on Uniform State Laws *Uniform Electronic Transactions Act, with Prefatory Note and Comments*. July 23-30, 1999, National Conference of Commissioners on Uniform State Laws: 211 E. Ontario Street, Suite 1300, Chicago, IL 60611. http://www.uetaonline.com.

Appendices

Appendix A: Sample ontology files

File format

Fields appear one per line, separated from their properties by commas.
Field order:

```
Field_name, data_type, text_description
```

Legal data types are:

string, double, integer, bytearray, boolean

Root ontology

The root ontology file `aexroot.ont` is automatically incorporated into every new atom. It provides fields that are required throughout the Atomic Market. Its field names are delimited with a percent-sign as the first character. No other ontology may name its fields with a percent sign in the first position, so name collisions are precluded.

```
%ontology,string, the name of this ontology
%description,string, describes the contents of this atom – a label for people
%quantity,double, how many?
%action,string, legal actions are OFFER and ASK
%importance,double, relative importance of item
%identity,string, this atom's creator's identity
```

Ontology for a book

```
title,string
isbn,string
upc,string
author,string
yearpublished,string
publisher,string
pages,integer
editor,string
condition,string
genre, string
```

Appendix B: The agent-config file

The agent-config file resides in the same directory as the generic agent, and provides several configuration values that control the runtime behavior of the agent. Usually, all agents belonging to a single entity will have mostly-identical entries in their configuration files, with the exception of the agents' individual names. The entries in an agent-config file, one per line, consist of a parameter name and a value, separated by a single comma. Blank lines and lines that begin with a "#" or semicolon are ignored by the parser.

The annotated agent-config file for the book-buying agent "buybot1" is presented below:

```
# Agent-config for BUYBOT1 agent
# ROLE -- most important, controls what 'role' files are loaded by this agent
role,bookbuyer  ← controls which strategy/compliance/evaluation files are loaded

# Communications
SMTP,smtp.newmediagroup.com          ← SMTP server for outgoing messages
POP3,pop3.newmediagroup.com          ← POP server for incoming messages
username,buybot1                     ← username for message retrieval
password,IMABOT                      ← password for message retrieval
myaddress,buybot1@newmediagroup.com  ← full address of agent

maxqueuesize,200        ← maximum number of messages to queue locally
polldelaymsec,1000      ← msec to delay between polling for incoming message
loopdelaymsec,1000      ← msec to delay main agent loop between iterations

                        (to avoid throttling the local machine; and for demos)

recirctimeout,60000     ← default time a message may recirculate locally before
                        it's killed (e.g. when advertising for trading partners)

recircmaxtraders,20     ← n=maximum trader to accept when advertising to fill
                        an incomplete component

nochangelimit,10        ← max hops to tolerate w/ no changes before dropping msg
hopcountlimit,20        ← max hops to allow before dropping message

ubermonitoraddress,monitor@newmediagroup.com   ← address of agent running the
                                               overview monitor, if any
```

```
# optional sound files, must be 8kHz .au files
tcsound,../sounds/gong.au                          ← transaction candidate declared
invalidpersonalsignaturesound,../sounds/sub_dive_horn.au  ← Alert! Invalid signature


# ***** REGISTRY SETTINGS USED BY THIS AGENT
# uncomment only one registryaddress and registrydriver
registryid,theregistry                ← for SQL servers, this is the name of the database
# if using the IDB local SQL database (for testing only, it is not multiuser):
#registryaddress,jdbc:idb:registrydb.prp
#registrydriver,org.enhydra.instantdb.jdbc.idbDriver
# when using MYSQL at aex.media.mit.edu:
registryaddress,jdbc:mysql://aex.media.mit.edu/theregistry
registrydriver,org.gjt.mm.mysql.Driver
registrymode,fullmetaljacket          ← make this 'local' to prevent notices going out


# key files
publickeyfile,publickey               ← where the agent s public key is stored
privatekeyfile,privatekey             ← where the agent s private key is stored
keycachefile,publickeycache           ← where the agent caches other agents public keys

strategydir,strategies                ← directory where strategy files are stored
complianceext,.com                    ← extension on Compliance rule files
evaluationext,.eva                    ← extension on Evaluation rule files
atomext,.atom                         ← extension on Atoms I respond to files
ontology-seekext,.oseek               ← extension on Ontologies I will seek files
commontermsext,.cterms                ← extension on Common Terms rule files
ontologytradingpartnersmaxsize,400    ← max trading partners to remember, per ontology
```

Appendix C: Reserved words

These keywords are sent as metadata with messages handled by the `Communications.sendMessage()` methods. With the exception of *BROADCAST*, these keywords are simply determined from the type of the transmitted object, and are presently used only for diagnostics:

MARKETMESSAGE

CONJUNCTIONNODE

ATOM

INTEGER

PING

PONG

MESSAGE

BROADCAST

Appendix D: National Weather Service raw data

The data returned by a lookup of Boston-area weather at the National Weather Service (NWS) arrives in the format show below. The project was fortunate to use Boston as an example; not all NWS forecasts use an easily-parsed tabular form. For sake of brevity, the listing has been abridged and blank lines removed, to better highlight relevant data.

```
<TT><PRE>FPUS61 KBOX 230802
SFPMA
MAZ002>024-001-CTZ001>012-RIZ001>007-232002-
STATE FORECAST FOR SOUTHERN NEW ENGLAND...UPDATED
NATIONAL WEATHER SERVICE TAUNTON MA
402 AM EDT MON APR 23 2001
ROWS INCLUDE...
        DAILY PREDOMINANT DAYTIME WEATHER
        FORECAST TEMPERATURES...DAYTIME HIGH...NIGHTTIME LOW
                - INDICATES TEMPERATURES BELOW ZERO
        DAYTIME PROBABILITY OF PRECIPITATION
        -- INDICATES MISSING VALUES

                  FCST     FCST     FCST     FCST     FCST
                  MON      TUE      WED      THU      FRI
                  APR 23   APR 24   APR 25   APR 26   APR 27
CITY
CONNECTICUT..
BRIDGEPORT, CT    PTCLDY   PTCLDY   PTCLDY   PTCLDY   PTCLDY
                  71/53    73/50    60/40    62/43    67
                  POP 0    POP 30   POP 10   POP 0    POP 0

MASSACHUSETTS-EASTERN..
BOSTON, MA        PTCLDY   PTCLDY   PTCLDY   PTCLDY   PTCLDY
                  65/51    78/52    57/41    59/44    64
                  POP 0    POP 30   POP 20   POP 20   POP 30
HYANNIS, MA       PTCLDY   MOCLDY   PTCLDY   PTCLDY   PTCLDY
                  61/50    68/52    56/41    58/46    63
                  POP 0    POP 20   POP 20   POP 20   POP 30
```

Appendix E: Source code for a skeleton XStrategy class

This example adds a book called “Three Little Pigs” to any market message presented to it. This is not at all useful, but it does illustrate how to create a class file that exploits the generic agent’s extension architecture.

To cause a generic agent to run this code, just place the XStrategy.class file in any generic agent’s home directory. Whenever the Strategy phase of message evaluation is run, this class will be loaded and executed. As the class is loaded dynamically, it may be replaced without requiring a recompilation or restart of the generic agent.

```
import edu.mit.media.jim.aexcore.*;

public class XStrategy {

        public XStrategy()
        {
        }

        static public void main(MarketMessage m, String _id)
        {
                new XStrategy();

                System.out.println("Tampering with MarketMessage");

                Atom x = new Atom("book");
                x.set("%identity","buybot1@newmediagroup.com");
                x.set("%quantity",12);
                x.set("title","Three Little Pigs");

                m.resetTraversal();
                AEXComponent c = m.nextBreadthFirstAEXComponent();
                c.insert("","AND",(new AEXComponent(x)));
        }
```