AGENT PROTOCOLS FOR PEER-TO-PEER ARCHITECTURES

Christopher D. Walton (cdw@inf.ed.ac.uk) [a*]

[a] *Centre for Intelligent Systems and their Applications (CISA), School of Informatics, The University of Edinburgh, Scotland, UK.*

Abstract

In this paper we present a technique which enables agents to participate in peer-to-peer (P2P) systems, such as file-sharing networks. Our technique is founded on the definition of lightweight protocols which specify the interactions required by the agent for a specific P2P network. The protocols that we define are executable specifications and can be directly implemented and independently verified. We present a definition of our MAP language for expressing protocols, and show how it can be used to enable participation in a simple P2P file-sharing system.

1 Introduction

A peer-to-peer (P2P) architecture is one which allows autonomous peers of similar capabilities to interact in a distributed and decentralised manner. The advantage of the P2P approach, over a centralised client/server architecture, is that the network resources are effectively utilised. This in-turn yields a more scalable and robust model of communication. P2P architectures have recently gained significant popularity for the distribution and sharing of files over the Internet. However, the potential scope for P2P techniques is much greater, and they can be effectively used in a range of different domains, including the Semantic Web, Grid Computing, Database Systems, and Multi-Agent Systems.

In many ways, P2P architectures are similar to Web-based Multi-Agent Systems (MAS). In particular, there are many conceptual similarities between peers and agents [5, 8]. However, P2P systems are generally assumed to involve very large numbers of participants, and require rapid communication between the network nodes. Therefore, multi-agent techniques which rely on undecidable reasoning, or lengthy theorem-proving operations are unsuitable for use in P2P systems. Instead, P2P systems rely on decidable and practical reasoning techniques which can be straightforwardly utilised by distributed peers. This eliminates many agent-oriented techniques from use in P2P systems, e.g. BDI reasoning, or run-time planning operations. However, many of the techniques for inter-operability and coordination within MAS are applicable and can be usefully employed in P2P architectures. It is our belief that there are significant benefits that can be realised by using MAS techniques in-tandem with P2P networks. We list four potential advantages below:

1. A P2P network can provide a platform on which to implement a MAS. The network is used as a facility for inter-agent communication.

2. A P2P network can act as a place from which agents can store and retrieve knowledge. The network is used as a knowledge base for one or more agents.

3. A P2P network can supply an agent or a group of agents with information. The network provides the environment in which the agents can execute.

4. An agent or group of agents can act as a bridge between different P2P networks.

*This work is sponsored by the EPSRC Advanced Knowledge Technologies (AKT) Interdisciplinary Research Collaboration (Grant GR/N15764/01).

To realise these benefits it is necessary for an agent to be able to act as either a client for a P2P network, or as a node which is fully part of the network.

One of the consequences of the rapid proliferation of P2P networks is that there are many competing standards and architectures. In particular, new techniques are frequently proposed to overcome architectural limitations, e.g. scalability issues, and structural compromises, e.g. loss of anonymity. These rapid changes make its difficult to design agents which can inter-operate with P2P networks reliably over a period of time. Typically, we would have to re-engineer our agents as new standards are developed, and produce specialised versions of our agents to operate on different networks. This situation is illustrated in Figure 1.

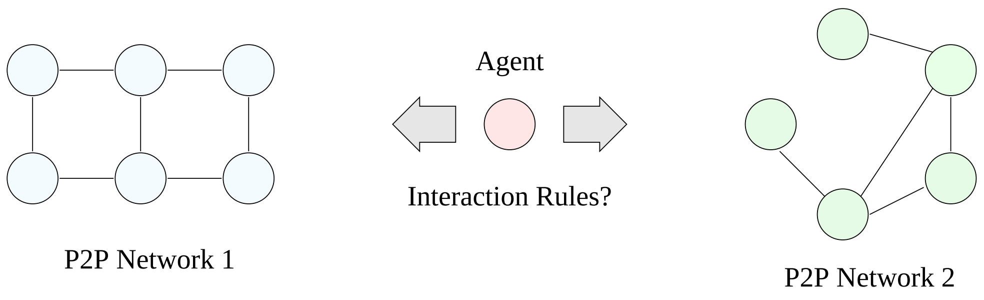


Figure 1: Multiple P2P Architectures.

From our preceding discussion, we would ideally like to engineer agents that can inter-operate with many different P2P networks, and thereby fully realise the benefits of the P2P approach that we have highlighted. However, in order to provide this level of inter-operability, we need to abstract away from any specific technology and define techniques that are applicable in general. Fortunately there are two key observations that can assist us in this task:

1. Most P2P networks are fundamentally the same at an abstract level. That is, P2P networks are generally a means to route information from one peer to another in a decentralised manner.

2. There is no conceptual difference between inter-agent communication, and communication between an agent and a P2P network. That is, in both cases there are specific rules that need to be followed, and underlying assumptions that must be obeyed.

These observations help us in the following ways. Firstly, we can identify common operations which can be used to describe P2P interactions at an abstract level. Secondly, we can utilise existing agent inter-operability techniques to interact with P2P networks.

The approach that we take in this paper is to define protocols which specify how to interact with a specific P2P network. These protocols define the rules of the network as a sequence of steps that the agent must follow. This approach is similar to the definition of social norms for agent communication in Electronic Institutions [1] and Conversation Policy [2]. The difference in our approach is that our specifications are directly executable by the agents, and our protocols are defined in terms of common P2P operations. Thus, and agent can interact with a new P2P network simply by obtaining a protocol specification for the network.

Our protocol language is defined as a lightweight formalism which specifies only the essential features of the communication process. This work is directly adapted from our previous work on interaction protocols in Multi-Agent Systems [12, 10]. However, the application to P2P networks in this paper is new. The focus of the paper is on the definition of the Multi-Agent Protocols (MAP) formalism for the realisation of interaction between agents in a P2P architecture. We present a script-based representation for the interaction between agents, which is lightweight and verifiable. Our approach has some similarities to the work in [6] which defines a formalism based on petri-nets for coordinating BDI agents in a P2P architecture. However, our approach is based on process calculus [3], and is not restricted to one specific model of agency.

Our presentation in this paper is structured as follows. In section 2 we define the MAP language for specifying an enacting protocols. To demonstrate the key features of the language, we present the specification of an example P2P file-sharing protocol in Section 3. Lastly, we conclude in Section 4 with a discussion of our future work.

2 MAP Language Definition

The MAP protocol language which we present here is a lightweight protocol language derived from process calculus, specifically the π calculus [4]. MAP is also derived from our previous work on multi-agent protocols, and thus we will use the terms *peer* and *agent* interchangeably. For convenience, we make the assumption that the internals of the agents are defined as a set of decision procedures, which we represent as a service. That is, we can describe our agents as a service-oriented architecture (SOA), with a specific interface for each agent.

MAP protocols can be viewed as *executable specifications*, and we have defined an execution framework for MAP, called MagentA [11]. Two key concepts in MAP are the division of protocols into *scenes*, and the assignment of *roles* to the peers. A scene can be thought of as a bounded space in which a group peers interact on a single task. Thus, a scene divides a large protocol into manageable parts. Scenes also add a measure of security to a protocol, in that peers which are not relevant to the protocol are excluded from the scene. This can prevent interference with the protocol and limits the number of exceptions and special cases that must be considered in the design of the protocol. We assume that a scene places a barrier on the peers, such that a scene cannot begin until all the peers have been instantiated.

The concept of a *role* is also central to our definition. In MAP, each peer is identified by both a name and a role. Peers are uniquely named, but must be assigned a role which is specified in the protocol. The role of an peer is fixed until the end of a scene, and determines which parts of the protocol the peer will follow. Peers can share the same role, which defines them as having the same capabilities, i.e. the same interface. Roles are useful for grouping similar peers together, as we do not have to specify a completely separate protocol for each individual. For example, we may wish to interact with a large number of peers, all with the same interface. We can simply define a single role (and associated protocol) which corresponds to the interface, rather than defining a separate protocol for each peer. Roles also allow us to specify multi-cast communication in MAP. For example, we can broadcast messages to all peers of a specific role.

We note that MAP is only intended to express protocols, and is not intended to be a general-purpose programming language. Therefore, the relative lack of features for performing computation is appropriate. Furthermore, MAP is designed to be a lightweight language and only a minimal set of operations have been included. It is intended that MAP protocols will be automatically generated, e.g. from a planning system. Thus, although MAP protocols appear complex, they would not generally be constructed by hand.

$$
\begin{array}{llll}
P \in \text{Protocol} & ::= & n(r\texttt{\{}\mathcal{M}\texttt{\}})^{+} & \text{(Scene)} \\
M \in \text{Method} & ::= & \texttt{method}\ m(\phi^{(k)})\ \texttt{=}\ op & \text{(Method)} \\
op \in \text{Operation} & ::= & \alpha & \text{(Action)} \\
 & \mid & op_1\ \texttt{then}\ op_2 & \text{(Sequence)} \\
 & \mid & op_1\ \texttt{or}\ op_2 & \text{(Choice)} \\
 & \mid & \texttt{waitfor}\ op_1\ \texttt{timeout}\ op_2 & \text{(Iteration)} \\
 & \mid & \texttt{call}\ m(\phi^{(k)}) & \text{(Recursion)} \\
\alpha \in \text{Action} & ::= & \epsilon & \text{(No Action)} \\
 & \mid & \phi^{k}\ \texttt{=}\ p(\phi^{l})\ \texttt{fault}\ \phi^{m} & \text{(Procedure)} \\
 & \mid & \rho(\phi^{(k)})\ \texttt{=>}\ \texttt{agent}(\phi_1,\ \phi_2) & \text{(Send)} \\
 & \mid & \rho(\phi^{(k)})\ \texttt{<=}\ \texttt{agent}(\phi_1,\ \phi_2) & \text{(Receive)} \\
\phi \in \text{Term} & ::= & \_ \mid a \mid r \mid c:\tau \mid v:\tau & \\
\tau \in \text{Type} & ::= & \mathit{utype} \mid \mathit{atype} \mid \mathit{rtype} \mid \mathit{tname} &
\end{array}
$$

Figure 2: MAP Abstract Syntax.

We will now define the abstract syntax of MAP, which is presented in Figure 2 (BNF notation). We have also defined a corresponding concrete XML-based syntax for MAP which is used in our MagentA implementation. However, we restrict our attention in this paper to the abstract syntax for readability. A protocol P is uniquely named n and defined as a set of roles r, each of which defines a set of methods $\mathcal{M}$. A method m takes a list of terms $\phi^{(k)}$ as arguments (the initial method is named `main`). Agents (i.e. peers) have a fixed role r for the duration of the protocol, and are individually identified by unique names a. Protocols are constructed from operations *op* which control the flow of the protocol, and actions α which have side-effects and can fail. Failure of actions causes backtracking in the protocol.

The interface between the protocol and the service which defines its behaviour, is achieved through the invocation of procedures p. A procedure is parameterised by three sequences of terms. The input terms ϕ^l are the input parameters to the procedure, and the output terms ϕ^k are the output parameters, i.e. results, from the procedure. A procedure may also raise an exception in which case the fault terms ϕ^m are bound to the exception parameters, and backtracking occurs in the protocol. Interaction between agents is performed by the exchange of messages which are defined by performatives ρ, i.e. message types. The parameters to procedures and performatives are terms ϕ, which are either variables v, agent names a, role names r, constants c, or wild-cards `_`. Literal data is represented by constants c in our language, which can be complex data-types, e.g. currency, flat-file data, multimedia, or XML documents. Variables are bound to terms by unification which occurs in the invocation of procedures, the receipt of messages, or through recursive method invocations. Constants and variables are assigned explicit types τ to ensure that they are treated consistently. We have previously presented a formal semantics of the MAP language in [10].

3 Gnutella Example

It is helpful to consider an example scenario in order to obtain an understanding of the MAP language and its application to P2P networks. The model that we describe is based on the Gnutella file sharing protocol. This protocol defines a completely decentralised method of file sharing, and its implementation is very straightforward. The Gnutella system assumes a distributed network of nodes, i.e. computers, that are willing to share files on the network. The protocol is defined with respect to our own node on the network, which we call the client. There are just three main operations performed by a client in the Gnutella protocol:

1. In order to participate in file sharing it is necessary to locate at least one other active node in the Gnutella network. There are a variety of ways in which this can be accomplished. The most common way is to contact one of more *Gwebcache* servers which which contain lists of recently active nodes. The Gnutella software is usually pre-configured with the addresses of a large number of these servers. However, it is not enough simply to know about other nodes, as there are no guarantees that these nodes will still be active. Therefore, the client will initiate a simple ping/pong protocol with each node in the list until a certain quota of active nodes have been located. This protocol simply sends a message (ping) to each node in the list, and waits for a certain period of time until a reply message (pong) is received, indicating that the node is still active.

2. Once a list of active nodes has been obtained, it is possible to perform a search for a particular file. Gnutella uses a *query flooding* protocol to locate files on the network. The client sends the file request (query) to every node on its active list. If one of the nodes has a copy of the requested file, then it sends back a reply message (hit) to the client. If the node does not have the file, then the request is forwarded to all of the active nodes on its own list, and so on. The query will eventually propagate to all of the nodes on the network, and the reply will be returned to the client.

3. If the file is successfully located by the query protocol, then then client simply contacts the destination node directly and initiates the download. If more than one copy is located, the client may download fragments of the file from different locations simultaneously and thereby improve download performance.

The Gnutella *ping/pong* and *query/hit* protocols are illustrated in Figure 3. It should be noted that the basic query flooding protocol, as outlined here, is very inefficient in operation and a search will typically take a long time to complete. The number of messages required is exponential in the depth of the search. This behaviour is tolerated as the network traffic generated by the queries is very small, compared to the bandwidth required to transfer the file itself. A variety of caching

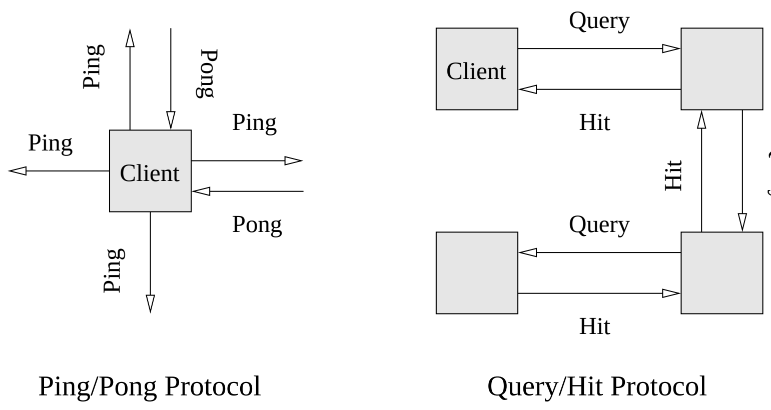


Figure 3: Query-flooding Protocol.

strategies have been proposed to improve the speed of the search, though we only consider the basic protocol here.

We can readily express the Gnutella protocol in MAP. The agents follow the protocol to determine the actions that must be performed by the nodes to retrieve a particular file. The encoding is presented in Figure 5 for the file-sharing nodes, and Figure 4 for an external client. We distinguish between the different types of terms by prefixing variables names with `$`, and role names with `%`. We use type abbreviations `a` for an agent, `r` for role, and `alist` for a list of agents.

```
1 %client{
2   method main() =
3     $node:id = getStartNode() then
4     $fname:string = getQuery() then
5     query($fname:string) => agent($node:a, %node) then
6     waitfor (hit($fname:string, $hitid:a) <= agent($name:a, $role:r))
7       then download($fname:string) => agent($hitid:a, %node) then
8       waitfor (filereply($file:file) <= agent($hitid:a, %node))}
```

Figure 4: MAP encoding of a P2P client.

The protocol for a node, shown in Figure 5, proceeds as follows. Upon initialisation (line 3), a list of neighbouring nodes is obtained, and a ping message is sent to all of these nodes in turn (lines 5-8). The node then enters a responsive state where is listens for incoming messages, and acts according to the message type. An incoming ping message results in an outgoing pong message (line 11). An incoming pong message is recorded in the list of active nodes (line 12). An incoming query results in an outgoing hit message if the node has a copy of the file (lines 13-15), or the query is forwarded to all of the neighbouring nodes (lines 16-18). An incoming hit message (from a neighbour) is forwarded to the initial requester of the query (lines 19-21). Finally, a download request message results in an outgoing message containing a copy of the file (lines 22-24). The protocol repeats after each message (line 25). For brevity, the `sendquery` and `sendhits` methods have been omitted as they have a similar definition to the `sendping` method. The procedures `startSharing`, `addActive`, `getActiveNodes`, `recordQuery`, `getQueryList`, and `getFile` are internal to the peer. We assume that these are standard P2P operations that the peer can provide. The client protocol shown in Figure 4 interacts directly with the node protocol that we have defined. A client obtains a node on the network (line 3), and constructs a query (line 4). The query is forwarded to the node (line 5), and the client waits for a hit message to be returned (line 6). A download is then initiated from the node which has a copy of the file (lines 7-8).

Our MAP protocols are clearly a straightforward implementation of the required functionality. However, there are some subtle issues that require further explanation. The operations in the protocol are sequenced by the `then` operator which evaluates op_1 followed by op_2, unless op_1 involved an action which failed. The failure of actions is handled by the `or` operator. This operator is

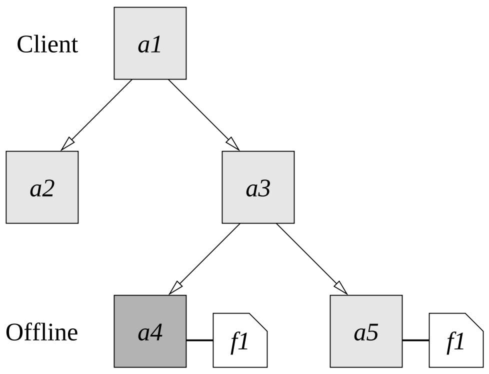
```
1 %node{
2   method main() =
3     $id:a = getId() then startSharing($id:a) then $nodes:alist = getNodes() then
4     (call sendping($nodes:alist) or call mainloop($id:a))
5   method sendping($nodes:alist) =
6     $head:a = Head($nodes:alist) fault nohead then
7     $tail:alist = Tail($nodes:alist) fault notail then
8     ping() => agent($head:a, %node) then call sendping($tail:alist)
9   method mainloop($id:a) =
10    waitfor
11         ((ping() <= agent($n:a, %node) then pong() => agent($n:a, %node))
12      or ((pong() <= agent($n:a, $role:r) then addActive($n:a, $role:r))
13      or ((query($f:string) <= agent($n:a, $r:r) then
14           ($fl:file = getFile($f:string) fault nofile then
15            hit($f:string, $id:a) => agent($n:a, $r:r))
16        or (setQuery($f:string, $n:a, $r:r) then
17            $nodes:alist = getActiveNodes() then
18            call sendquery($f:string, $nodes:alist)))
19      or ((hit($f:string, $hid:a) <= agent($n:a, %node) then
20          $nodes:alist = getQueryList($f:string) then
21          call sendhits($f:string, $hid:a, $nodes:alist))
22      or (download($f:string) <= agent($client:a, %client) then
23          $fl:file = getFile($f:string) fault nofile then
24          file($fl:file) => agent($client:a, %client))))))
25    then call mainloop($id:a)}
```

Figure 5: MAP encoding of a P2P node.

defined such that if op_1 fails, then op_2 is evaluated, otherwise op_2 is ignored. The language includes backtracking, such that the execution will backtrack to the nearest `or` operator when a failure occurs. Similarly, the body of a `waitfor` loop will be repeatedly executed upon failure, and the loop will terminate when the body succeeds.

The semantics of message passing in MAP corresponds to non-blocking, reliable, and buffered communication. Sending a message will succeed immediately if an agent matches the definition, and the message will be stored in a buffer on the recipient. When exchanging messages through send and receive actions, a unification of terms against the definition `agent(`ϕ_1`,` ϕ_2`)` is performed, where ϕ_1 is matched against the agent name, and ϕ_2 is matched against the agent role. For example, the receipt of the ping message in line 11 of the node protocol will match any agent whose role is `%node`, and the name of this node will be bound to the variable `$n`. In this definition, a client is not permitted to send a ping message to a node. Although not illustrated in this example, we can use a wild-card `_` to send a message to all agents regardless of their role. The advantage of non-blocking communication is that we can check for a number of different messages at the same time. Race conditions are avoided by wrapping all receive actions by `waitfor` loops. A `waitfor` loop can also include a `timeout` condition which is triggered after a certain interval has elapsed.

To illustrate the execution of the MAP P2P protocol, we define an example file sharing network in Figure 6. The network is composed from six individual nodes, labelled $a1$ though $a6$, where $a1$ is the client. Each node is only aware of those directly connected in the graph, for example, $a1$ is aware only of $a2$ and $a3$, and $a5$ is only aware of $a6$. There is one file $f1$ which is contained on nodes $a4$ and $a5$. However, node $a4$ is currently off-line, i.e. unavailable. In this example, the intention of node $a1$ is to retrieve file $f1$. The sequence of interactions between nodes $a1$, $a3$, and $a5$ are shown in MAP syntax. For brevity, we have omitted the initial ping/pong process. Also, we only show the message exchanges between the nodes. The sequence of steps illustrates how the query is propagated from node $a1$ to node $a5$, and how the file download is performed.



Agent $a1$:
```
query(f1) => agent(a2, %node) then
query(f1) => agent(a3, %node) then
hit(f1, a5) <= agent(a3, %node) then
download(f1) => agent(a5, %node) then
filereply(f1) <= agent(a5, %node)
```

Agent $a3$:
```
query(f1) <= agent(a1, %client) then
query(f1) => agent(a4, %node) then
query(f1) => agent(a5, %node) then
hit(f1, a5) <= agent(a5, %node) then
hit(f1, a5) => agent(a1, %client)
```

Agent $a5$:
```
query(f1) <= agent(a3, %node) then
hit(f1, a5) => agent(a3, %node) then
download(f1) <= agent(a1, %client) then
filereply(f1) => agent(a1, %client)
```

Figure 6: Example P2P Network.

4 Conclusion

The purpose of this paper is to demonstrate that we can use protocols to allow agents to participate in P2P networks. In doing so, we enable our agents to realise the scalability and robustness benefits of these networks. Our approach is independent of any specific P2P technology, and thus we can construct agent which can interact with a range of different P2P networks without a difficult re-engineering process. Our technique is founded on MAP, which is a formally-defined and executable protocol language. MAP permits us to define the essential interactions with a P2P network in terms of common operations and communication patterns. Thus, all that an agent needs to interact with a new P2P network is an appropriate protocol specification. The agent can retrieve this specification and immediately participate in the network. The separation of the protocol from the agent also permits protocols to be externally verified, e.g. to ensure fairness and to eliminate deadlocks. We have previously shown how this can be accomplished by using model checking [9].

At the present time, the protocols must be constructed by hand. We acknowledge that this can be a difficult and time-consuming process, and therefore we are currently considering a number of approaches which will permit protocols to be constructed in a more efficient and/or automated manner. The most straightforward approach is the provision of a graphical tool for constructing protocols. Beyond this, we would like to support the automatic generation of protocols. We have made some initial progress into the construction of protocols as an outcome of a plan synthesis process. This process may also be used to synthesis new P2P protocols. We are also considering extending the MAP language with features that would make it more suited to P2P architectures, such as those based on Distributed Hash Tables (DHT) [7]. These enhancements include explicit support for different message communication patterns, improved fault-tolerance mechanisms, and additional data-types.

References

[1] M. Esteva, J. A. Rodríguez, C. Sierra, P. Garcia, and J. L. Arcos. On the Formal Specification of Electronic Institutions. In *Agent-mediated Electronic Commerce (The European AgentLink Perspective)*, volume 1991 of *Lecture Notes in Artificial Intelligence*, pages 126–147, 2001.

[2] M. Greaves, H. Holmback, and J. Bradshaw. What is a Conversation Policy? In *Proceedings of the Workshop on Specifying and Implementing Conversation Policies, Autonomous Agents '99*, Seattle, Washington, May 1999.

[3] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[4] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (Part 1/2). *Information and Computation*, 100(1):1–77, September 1992.

[5] M. Oriol. Peer Services: From Description to Invocation. In *Proceedings of the First International Workshop on Agents and Peer-to-Peer Computing (AP2PC02)*, volume 2530 of *Lecture Notes in Computer Science*, pages 21–32, July 2002.

[6] M. Purvis, M. Nowostawski, S. Cranefield, and M. Oliveira. Multi-agent Interaction Technology for Peer-to-Peer Computing in Electronic Trading Environments. In *Proceedings of the Second International Workshop on Agents and Peer-to-Peer Computing (AP2PC03)*, volume 2872 of *Lecture Notes in Computer Science*, pages 150–161, July 2003.

[7] R. Siebes. pNear: Combining Content Clustering and Distributed Hash Tables. In *Proceedings of the IEEE'05 Workshop on Peer-to-Peer Knowledge Management (P2PKM05)*, San Diego, CA, July 2005.

[8] M. Singh. Peer-to-Peer Computing for Information Systems. In *Proceedings of the First International Workshop on Agents and Peer-to-Peer Computing (AP2PC02)*, volume 2530 of *Lecture Notes in Computer Science*, pages 15–20, July 2002.

[9] C. Walton. Model Checking Multi-Agent Web Services. In *Proceedings of the 2004 AAAI Spring Symposium on Semantic Web Services*, Stanford, California, March 2004. AAAI.

[10] C. Walton. Multi-Agent Dialogue Protocols. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, January 2004.

[11] C. Walton and A. Barker. An Agent-based e-Science Experiment Builder. In *Proceedings of the 1st International Workshop on Semantic Intelligent Middleware for the Web and the Grid*, Valencia, Spain, August 2004.

[12] C. Walton and D. Robertson. Flexible Multi-Agent Protocols. In *Proceedings of UKMAS 2002. Also published as Informatics Technical Report EDI-INF-RR-0164, University of Edinburgh*, November 2002.