

# A distributed implementation of the SWAN peer-to-peer look-up system using mobile agents

Erwin Bonsma and Cefn Hoile

Intelligent Systems Lab, BTextact Technologies  
Austral Park, Ipswich, IP5 3RE, U.K.  
erwin.bonsma@bt.com cefn.hoile@bt.com

**Abstract.** We show how the SWAN system has been implemented using a multi-agent system. SWAN is peer-to-peer look-up system that functions by letting the participating nodes self-organise in a virtual Small World Network. The multi-agent system we are using is DIET, a light-weight ecologically inspired agent platform. We describe how the test application can be implemented, and present experiments in which the application runs on a cluster of computers. Our results show that the system is robust to failure and shows promising scalability.

## 1 Introduction

Recently there has been much interest in peer-to-peer applications, file sharing, file storage and collaborative working. Many of these applications require the ability to look-up items, ideally by a location-independent identity so that items can easily be moved and duplicated. How to implement a look-up system in peer-to-peer systems is an interesting question because ordinary, centralised solutions are typically not suitable. Various distributed look-up systems have been proposed, including Chord [10], Pastry [9], CAN [7] and SWAN [1].

Here we look at the SWAN look-up system. More specifically, we extend our initial work where we simulated the system [1] by using the DIET multi-agent system [6, 3] to build a test application that can run across multiple computers.

This paper is organised as follows. Section 2 and 3 provide the relevant background by respectively introducing the DIET platform and the SWAN system. Section 4 then describes the implementation of SWAN using DIET that is the subject of this paper. Some preliminary results about the performance of the system can be found in Section 5. Finally, Section 6 and 7 discuss the results and conclude the paper.

## 2 The DIET platform

The DIET (Decentralised Information Ecosystem Technologies) platform is a multi-agent platform in Java that is being developed as part of the DIET project. The DIET project is a 5th Framework project funded by the European Commission under the Future and Emerging Technologies area [2]. One of the main goals

of the project has been to design a multi-agent platform that is open, robust, adaptive and scalable, using a substantially bottom-up and ecosystem-inspired approach. As a result, the DIET platform differs significantly from most other multi-agent platforms.

*Environments* in DIET provide a location for agents to inhabit and can host one or more agents. DIET agents access the DIET kernel functionality by way of their environment. In DIET there is one *world* per Java Virtual Machine (JVM) which can contain one or more environments. Each environment can have *neighbourhood links* to other environments, which are not necessarily in the same world. These neighbourhood links allow agents without any a-priori knowledge of other environments to explore the DIET universe by migration. The neighbourhood links are specified on start up of the world, but can be changed dynamically while the world is running. There is no need for a central registration of worlds when you run DIET across multiple computers, so the architecture is effectively peer-to-peer.

Agents in DIET are very lightweight. Their minimal memory footprint is small and they can give up their thread when they do not need it. The DIET kernel will then attempt to give agents a thread when they need it, for instance to handle incoming messages. The lightweight nature has allowed us to run simulations with several hundred thousand agents in one JVM on a single, ordinary desktop computer.

For both communication and agent migration the kernel support is minimal. For instance, the platform has no specific communication protocol built-into it, remote communication is not directly supported and migration is unreliable. This means that there is no unnecessary overhead and execution can be rapid. However, if need be, more sophisticated functionality can be built on top of the basic functions provided by the kernel, in a way best suited to the conditions in which it is used.

Robustness is explicitly addressed in the DIET kernel by directly exposing agents to potential failure. The kernel only fulfills a request when it can easily do so, and fails when it cannot. For instance, local message delivery fails when the receiving agent cannot be assigned a thread, or its message buffer has reached full capacity. In this way, no extra resources are spent when the system is apparently already overloaded. Additionally, by exposing the agents to failure, they can adapt accordingly. For more details about the DIET platform refer to [6, 3].

### 3 The SWAN system

The SWAN (Small World Adaptive Networks) system provides distributed look-up functionality for peer-to-peer infrastructures. In SWAN it is assumed that a collection of virtual *nodes* is connected by a communication infra-structure, and that each node is given a potentially arbitrary location-independent *node identity*. The purpose of SWAN is to let nodes self-organise so that each node can be found given its identity. The architecture is fully decentralised and all nodes contribute equally to the system.

Nodes have three types of links: *bootstrap links*, *short-range links* and *long-range links*. The short-range and long-range links form a virtual *SWN* (Small World Network) that is used to find nodes. The bootstrap links are used to create the SWN. On start up, each node needs to be provided with a few random bootstrap links to other nodes. Initially nodes have no short-range and long-range links.

The nodes self-organise into a SWN that is based on their identities. Each node identity is mapped to a position in a multi-dimensional *identity space*. The SWN is created according to the distances in this identity space.

The short-range links are formed by letting nodes periodically send Notify messages. These are randomly passed along the bootstrap links to reach an arbitrary position in identity space. Subsequently, the message is routed back to the originating node, along the current SWN links. When the originating node is not reached, some nodes will update their links accordingly. Eventually, each node will have short-range links to other nodes that are nearby in identity space.

The second part of the self-organisation is the formation of long-range links. They are to nodes that are further away in identity space. Their distances are chosen such that the SWN meets the properties suggested by Kleinberg [4, 5], which guarantees that the average effort required to find a node scales well with the total number of nodes. Each node gradually improves their long-range links by issuing Find queries for identities that are at a desired distance from its own identity.

To summarise, self-organisation is achieved by passing many small messages. Every node periodically initiates a message, and routes incoming messages according to a simple protocol. These messages are initially responsible for creating the SWN network, and subsequently ensure that the network adapts to failure and handles the arrival and departure of nodes. The resulting network can be used to efficiently find any node, with an effort that scales well with the total number of nodes. Experiments have shown that the self-organisation process also has good scalability. More details about SWAN can be found in [1].

### 3.1 SWAN API

A prototype of the SWAN protocol has been implemented in Java. Most of the details of the protocol are hidden from applications that want to use SWAN. This is achieved by using the *SwanEngine* and *SwanEngineContext* interfaces shown below.

```
interface SwanEngine {  
  
    /* Sets up and activates the SWAN protocol engine.  
     */  
    void activate(SwanIdentity id, SwanEngineContext context,  
                 SwanAddress internal_address,  
                 SwanAddress external_address);  
}
```

```

    /* Provides the SWAN node with an additional bootstrap link.
     */
    void addBootstrapLink(SwanAddress address);

    /* Handles incoming SWAN messages.
     */
    void handleMessage(SwanMessage msg);

    /* Initiates a SWAN Find query.
     */
    void find(SwanIdentity target_id);
}

interface SwanEngineContext {

    /* Called by the SWAN engine to send outgoing SWAN messages.
     */
    void sendMessage(SwanMessage msg);

    /* Called by the SWAN engine to signal it wants a bootstrap
     * link.
     */
    void requestBootstrapLink();

    /* Called by the SWAN engine to return the result of a Find
     * query.
     */
    void findDone(SwanIdentity target_id, SwanIdentity subject_id,
                  SwanAddress subject_address);
}

```

The *SwanEngine* interface is implemented by a generic SWAN protocol engine. The *SwanEngineContext* interface must be provided by the object that hosts the protocol engine, and provides the protocol engine with a context to interact with the application. For instance, the protocol engine does not send any messages, but uses the context to do so. In this way, the application can use a message delivery mechanism that is most suited to it. The protocol engine also uses the context to request bootstrap links. The application can use the *SwanEngine* interface to issue Find queries, and the results are returned to it by way of the *SwanEngineContext* interface.

There are two addresses associated with the SWAN node, both are set when the SWAN protocol engine is activated. The *internal address* is where all SWAN messages are received. These are low-level messages that are used to create and maintain the SWN, and for handling Find queries. The *external address* on the other hand, is reserved for application-level messages. This address is returned

in response to external Find queries, and is therefore where any subsequent application specific messages are received.

## 4 Implementation of SWAN using DIET

To investigate how SWAN performs when it runs across multiple computers, we have implemented a test application in DIET. We have used DIET because its agent communication and migration capabilities provide a useful framework for developing distributed peer-to-peer applications. Furthermore, DIET's approach to robustness suits SWAN very well because SWAN does not require the reliable message delivery. Moreover, it is desirable in SWAN to quickly expose nodes to failure, so that the SWN can adapt as soon as possible. DIET's lightweight nature is also useful, because self-organisation in SWAN requires the sending of many, small messages.

At the heart of the application are the types of agent listed in Table 1.

**Table 1.** The different types of agents that are used

Type	Functionality
<i>ServiceProvider</i>	Makes itself accessible in the SWAN system
<i>SwanEngineManager</i>	Manages one or more SWAN protocol engines
<i>Carrier</i>	Performs basic remote communication
<i>BootstrapScout</i>	Finds suitable SWAN bootstrap links
<i>Tester</i>	Periodically samples the state of the SWAN network
<i>TargetFinder</i>	Used by the Tester to discover random Find targets

Multiple *ServiceProvider* agents are created. Each *ServiceProvider* makes itself accessible as a node in the SWAN system. On creation, each agent effectively generates a random node identity for itself. The agent also creates a SWAN protocol engine that is responsible for establishing the agent as a node in the SWAN network. The *ServiceProvider* agent uses the SWAN network to handle queries from other agents to find specific *ServiceProvider* agents. It does not offer any additional functionality here, but it would in a more realistic application. In a file-sharing application, for instance, each agent could be responsible for hosting a file and the identity of the SWAN nodes could be based on the filename. In this way, files can be retrieved by name, even after they moved to a different location.

The *SwanEngineManager* is the most advanced agent of all. It manages one or more SWAN protocol engines. On start up, each *ServiceProvider* hands over its SWAN protocol engine to a *SwanEngineManager* in its environment. From then on, the *SwanEngineManager* is responsible for receiving and sending all SWAN messages for these protocol engines. However, any messages related to the services provided by the *ServiceProviders* are sent to these agents directly.

In this way, communication with ServiceProvider agents is not affected by the SwanEngineManager, and agents that use the ServiceProvider can be fully unaware of the SwanEngineManager. This is achieved by setting the internal and external SWAN addresses for each node as shown in Table 2. The `local ID` is an identifier local to the SwanEngineManager, which is used to efficiently forward each incoming SWAN message to the appropriate SWAN protocol engine.

**Table 2.** The two different addresses per node in SWAN

Type	Address
internal	<SwanEngineManager address><local ID>
external	<ServiceProvider address>

The SwanEngineManager makes use of two types of agent: Carriers and BootstrapScouts. *Carrier* agents are part of the standard DIET platform. They can perform remote communication, i.e. send a message to an agent in a different environment. A Carrier does so by migrating to the remote environment, connecting to the target agent and delivering the message to it. *BootstrapScout* agents are specific to SWAN. They discover SWAN nodes in other environments, which are then used by the SwanEngineManager to provide each SWAN node with suitable bootstrap links. BootstrapScouts simply find these other nodes by performing a random walk, along environment neighbourhood links.

The locally-centralised management of SWAN protocol engines has several advantages. Firstly, ServiceProvider agents do not receive any SWAN messages. They can therefore easily give up their thread. If a ServiceProvider temporarily needs a thread, for instance when it is contacted by another agent, the DIET kernel attempts to give it one so that it can respond. This makes sense from a resource point of view: Why permanently give a thread to agents that are inactive most of the time? It also helps to scale up the number of ServiceProviders that can be hosted in a single JVM.

Secondly, use of SwanEngineManagers makes it straightforward to improve the remote communication mechanism that is used. Currently, Carrier agents are used to carry messages to agents in different environments. This uses agent migration, which is implemented using TCP sockets. In SWAN many messages are sent to potentially many different destinations, but message delivery does not need to be reliable. Therefore connection-less UDP would be more efficient here than reliable connection-based TCP. The SwanEngineManager can set up a UDP port to receive SWAN messages for all engines it is managing. It can forward each message to the appropriate engine, similar to how it is currently forwarding each message.

Note that the “centralisation” added by the SwanEngineManager does not negatively affect the robustness. The reason is that all agents reside in the same JVM, and thus share the same CPU, memory and network connection. If there

is a hardware failure, or the system is temporarily overloaded, all agents in the same environment are affected anyway, even if each ServiceProvider would manage its own SWAN protocol engine.

The task of the *Tester* agent is to give an indication of how the self-organisation is progressing, but it does not contribute to the self-organisation process itself. It periodically samples the quality of the SWAN network by issuing a Find request. It uses a *TargetFinder* agent to randomly find a target ServiceProvider agent. The TargetFinder does this by starting a random walk along neighbourhood links. In the environment where it ends, it randomly contacts one of the ServiceProviders and asks for its SWAN node identity. It then migrates back to its home environment where it reports back to the Tester with the identity. The Tester then contacts one of the local ServiceProviders and asks it to issue a Find query for the given identity to see if it can indeed be found.

Next to the agents, which make up most of the application, there are some monitoring and visualisation components. On start up, these attach themselves to environments, agents and SWAN protocol engines and are notified of relevant events. These components keep track of the total number of messages that are sent, the links that are currently maintained by a particular SWAN node, the state of the overall SWAN network, etc.

## 5 Running SWAN using DIET

We have ran the application on a Beowulf cluster [8] consisting of 18 computers, each with Dual Pentium 450 MHz CPUs. On each computer that we use, we run a single DIET world containing a single environment. On start up, each environment is filled with a single SwanEngineManager, a single Tester and multiple ServiceProviders. The other agents are created as and when they are needed. The Tester is configured such that it issues a Find query once every second. The swan protocol engines are configured so that the node activity leads to an overall CPU usage of approximately 50%.

Experiment 1 demonstrates how the system adapts to the arrival and departure of nodes. First we start up a single world, World 1, with one hundred ServiceProviders. After the SWN has sufficiently self-organised, we start up a second world, World 2, with another hundred ServiceProviders. Once the SWN has sufficiently self-organised again, we abruptly terminate World 2 and wait for the SWN to adapt, after which we terminate World 1. More precisely, we start up World 2 after ten subsequent find queries in World 1 have been successful. We terminate World 2 after in both worlds ten subsequent find queries have been successful. We finally terminate World 1 once ten subsequent find queries have been successful again.

Table 3 gives an idea of the time and effort required for the SWN to adapt. It shows at various moments during run, the total number of agents that were created and that arrived in World 1, as well as the total number of messages sent by these agents. Figure 1 and Figure 2 both show the SWN as seen from World 1, at different stages of the experiment. These figures only show the short-range

links, as the figures would be very cluttered otherwise. The shading of the links indicates the directionality of each link. When a link originates from a node, the corresponding end of the link is dark. The nodes in World 2 and the link that they are maintain are not drawn, which explains why there is not a node at the end of all links in Figure 2. Finally, Figure 3 shows a screenshot of the visualisation components. The components shows all the links maintained by a SWAN node in World 1. The address column shows that the node has several links to nodes in World 2, on a different computer.

The entire run took approximately five and a half minutes, which shows that the SWAN system can adapt fairly rapidly. In fact, the system was configured such that links would be removed after two subsequent messages were delivered successfully. Lowering this threshold to one failed message per link would let the SWN adapt even more quickly after World 2 is killed.

**Table 3.** The total time and total agent and message activity in World 1 at various stages of Experiment 1

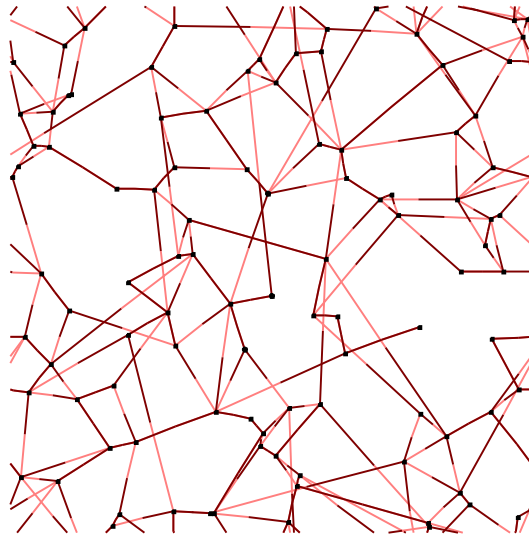
	Time [s]	Agents created	Agents arrived	Messages sent
Start of World 1	0	0	0	0
Start of World 2	60	467	0	1435
End of World 2	168	3278	3424	6162
End of World 1	327	4324	3424	7138

The aim of Experiment 2 is to see how the application performs when there are many nodes. We start a world on each of the eighteen computers in the cluster, and place 600 ServiceProviders in each world to create 10800 SWAN nodes in total. We run the application for six hours.

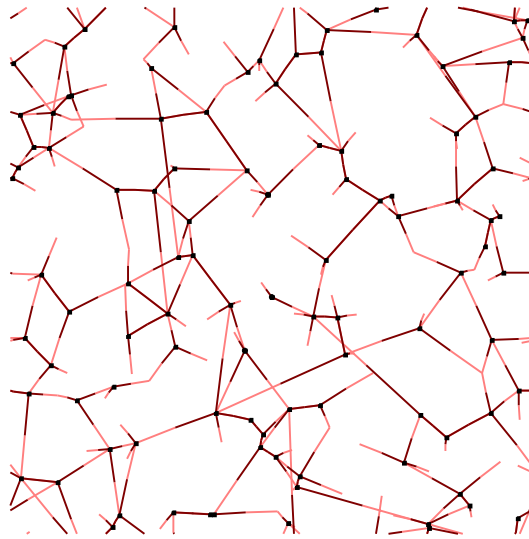
Figure 4 shows the results of the Find queries gathered by one of the Tester agents. The plots are generated by taking a sliding average over one hundred seconds. It can be seen that it takes thirty-five minutes before half of the Find queries succeeds. It is not surprising that the self-organisation takes longer than it did in Experiment 1. Not only did the total number of nodes increase from 200 to over 10000, the available CPU time per node also decreased by a factor six.

There are also five obvious dips in the performance. A closer examination of the experimental data explains these results. Occasionally, a DIET world can become overloaded. Due to a lack of available threads, no agents can be created or migrate into the world. In the fourth dip, the world that the Tester is running in happens to be overloaded. The Tester can therefore not create any TargetFinder agents, and as a result, does not issue any Find queries. The other dips are caused by failures in one of the other worlds.

What exactly causes these temporary world failures is yet unknown. It could be due to a bug in the application, in DIET or in SWAN. Alternatively, it could



**Fig. 1.** The short-range links in World 1 just before World 2 started in Experiment 1



**Fig. 2.** The short-range links in World 1 just before World 2 ended in Experiment 1

SWAN engine tracker					
[80e8][8d49]		[7ed2][5b24]		[7399][a735]	
Identity	Type	Created	Sent	Rec...	Address
	Boot	14:12:01	5	5	049bbf7ea-0fdf16d43@10.0.0.1:...
	Boot	14:12:02	8	8	049bbf7ea-0fdf16d43@10.0.0.1:...
	Boot	14:12:12	4	4	049bbf7ea-0fdf16d43@10.0.0.1:...
[623b][f729]	Long	14:12:16	2	2	049bbf7ea-0fdf16d43@10.0.0.1:...
	Boot	14:12:22	3	3	049bbf7ea-0fdf16d43@10.0.0.1:...
	Boot	14:12:32	5	5	049bbf7ea-0fdf16d43@10.0.0.1:...
	Boot	14:12:41	4	4	049bbf7ea-0fdf16d43@10.0.0.1:...
[7645][58ab]	Short-1	14:13:09	9	9	08f9dcf61-0fdf16d43@10.0.0.3:...
[7715][5811]	Short-2	14:13:13	13	12	08f9dcf61-0fdf16d43@10.0.0.3:...
[980a][42fb]	Short-1	14:13:27	9	9	049bbf7ea-0fdf16d43@10.0.0.1:...
[2e43][799f]	Long	14:13:37	1	1	08f9dcf61-0fdf16d43@10.0.0.3:...
[7750][660c]	Short-1	14:13:40	9	9	08f9dcf61-0fdf16d43@10.0.0.3:...
[9e79][5b0e]	Short-2	14:13:47	6	6	08f9dcf61-0fdf16d43@10.0.0.3:...
[6935][25f1]	Long	14:13:57	3	3	08f9dcf61-0fdf16d43@10.0.0.3:...
[6c76][9113]	Short-2	14:14:17	3	3	049bbf7ea-0fdf16d43@10.0.0.1:...
[8b89][952a]	Long	14:14:36	0	0	08f9dcf61-0fdf16d43@10.0.0.3:...

Fig. 3. Visualisation component track the links for one or more SWAN nodes. The screenshot it taken just before World 2 ended in Experiment 1

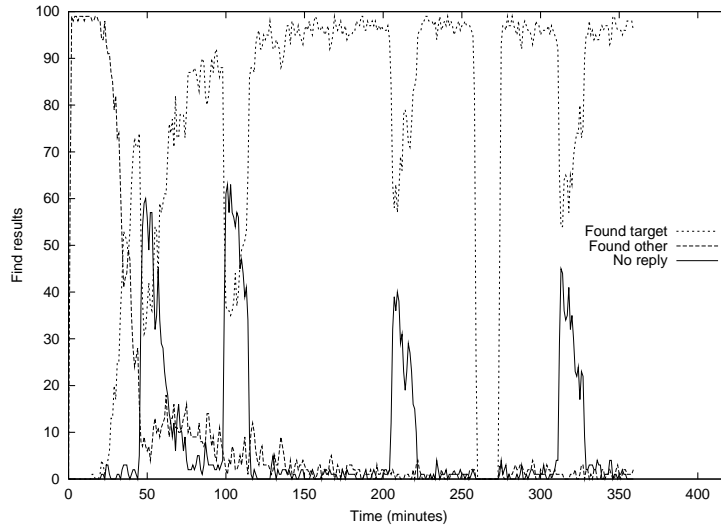


Fig. 4. The Find query results in Experiment 2 gathered by one of the Tester agents

be due to a problem in the standard Java libraries, the JVM we are using, or a problem with sockets or multi-threading in the Operating System. Irrespective of the cause, it is very encouraging that that the application keeps on functioning. Considering that 600 SWAN nodes become unaccessible at once, the dip in the performance is relatively small, and after a while the application recovers entirely.

## 6 Discussion

We found that implementing the system in DIET was advantageous in several ways. Firstly, we think that an agent-based approach in general is a useful paradigm for developing prototype peer-to-peer systems. The autonomous execution of agents with their ability to interact, makes it straightforward to implement peer-to-peer applications, where possibly there are multiple peers per computer.

Secondly, the lightweight nature of DIET was ideal for the application presented here. SWAN relies on the passing of many small messages, so it is important that this can be done efficiently. Additionally, the fact that agents do not permanently need to have a thread assigned to them has helped to increase the number of nodes that could be supported.

Thirdly, the robustness of DIET has been important. When the application is ill-configured, or due to temporary glitches, parts of it can become overloaded. If DIET would try to ensure that all messages were handled anyway, this would inevitably cause the entire system to grind to a halt as more and more resources are needed to store and deliver pending messages. However, DIET can discard messages and agents such that the system as a whole can keep on functioning. This only works properly if the application does not rely on perfect message delivery, but this is a sensible design approach for peer-to-peer systems anyway.

There is still plenty of scope for future work. Firstly, the experiments presented here have been limited. We are planning to perform additional experiments. It would be useful to generate more quantitative results, which for instance show how the number of nodes affects the speed of the self-organisation process. Additionally, it would be useful to test the adaptability and robustness of the system, by constantly introducing new nodes, removing arbitrary existing nodes, and changing the location of nodes.

Secondly, the system presented here can be enhanced in various ways. For instance, as mentioned in Section 4, the `SwanEngineManager` agent could be extended to use UDP for delivering messages. This would reduce the overhead associated with sending each message, and also reduces the number of sockets that is needed per JVM. The latter becomes important when you want to run the application on many more machines.

Thirdly, the application should of course be extended to do something “useful” to the user. We are currently using the software described in this paper to build several demo applications.

## 7 Conclusion

We have presented a distributed implementation of the SWAN look-up system using the DIET multi-agent platform. DIET's approach to robustness and its lightweight nature made it very suitable for running SWAN. An important feature of SWAN is that it is robust to failure, which has been demonstrated by the experiments. The experiments also show that the system is able to self-organise relatively quickly when it is run across multiple computers. The fact that ten thousand SWAN nodes could be hosted on only eighteen computers is also encouraging.

## References

1. Erwin Bonsma. Fully decentralised, scalable look-up in a network of peers using small world networks. In *Proc. 6th Multi Conf. on Systemics, Cybernetics and Informatics (SCI2002)*, Orlando, July 2002.
2. European Commission IST Future and Emerging Technologies. Universal information ecosystems proactive initiative. <http://www.cordis.lu/ist/fethome.htm>, 1999.
3. Cefn Hoile, Fang Wang, Erwin Bonsma, and Paul Marrow. Core specification and experiments in DIET: A decentralised ecosystem-inspired mobile agent system. In *Proc. 1st Int. Conf. on Autonomous Agents and Multi-Agent Systems (AAMAS2002)*, Bologna, Italy, July 2002.
4. Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. Technical Report 99-1776, Department of Computer Science, Cornell University, Ithaca NY 14853, October 1999.
5. Jon M. Kleinberg. Navigation in a small world. *Nature*, 305:845, August 2000.
6. P. Marrow, M. Koubarakis, R.H. van Lengen, F. Valverde-Albacete, E. Bonsma, J. Cid-Suerio, A.R. Figueiras-Vidal, A. Gallardo-Antolín, C. Hoile, T. Koutris, H. Molina-Bulla, A. Navia-Vázquez, P. Raftopoulou, N. Skarmas, C. Tryfonopoulos, F. Wang, and C. Xiruhaki. Agents in decentralised information ecosystems: the DIET approach. In *Proc. of the AISB01 Symposium on Information Agents for Electronic Commerce*, pages 109–117, York, UK, 2001.
7. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, San Diego CA, August 2001.
8. Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas Sterling. Beowulf: Harnessing the power of parallelism in a pile-of-PCs. In *Proc. IEEE Aerospace*, volume 2, pages 79–91, 1997.
9. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Int. Conf. on Distr. Sys. Platforms*, Heidelberg, Germany, November 2001.
10. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, San Diego CA, August 2001.