A Scalable and Secure Global Tracking Service for Mobile Agents

Volker Roth and Jan Peters

Fraunhofer Institut für Graphische Datenverarbeitung
Rundeturmstraße 6, 64283 Darmstadt, Germany
{vroth|jpeters}@igd.fhg.de

Abstract. In this paper, we propose a global tracking service for mobile agents, which is scalable to the Internet and accounts for security issues as well as the particularities of mobile agents (frequent changes in locations). The protocols we propose address agent impersonation, malicious location updates, as well as security issues that arise from profiling location servers, and threaten the privacy of agent owners. We also describe the general framework of our tracking service, and some evaluation results of the reference implementation we made.

Keywords: mobile agents, tracking, agent name, security.

1 Introduction

Early research on mobile agent systems concentrated on how to migrate mobile agents. Many agent systems exist today and there is a notable shift towards research in how to best support *transparent communication* between mobile agents [20, 16, 10, 11]. *Transparent* means that agents need not be aware of the actual location of agents with which they wish to communicate.

Two general problems must be solved in order to achieve transparent communication: first, the peer agent must be located, for instance by means of a *tracking service* that maps an agent's location invariant name onto its current whereabouts. Second, messages must be routed to the peer. This can become difficult since mobile agents might "run away" from messages. Guaranteed delivery is addressed for instance by Murphy, Picco, and Moreau [11, 10].

In this article we address the problem of establishing a public global tracking service for mobile agents, which scales to the Internet and accounts for the particularities of mobile agents (frequent changes in locations). *Public* means that lookups of agent locations are not restricted in principle, yet we would like to account for security and privacy issues arising for agent owners in using such a tracking service. More precisely, the following problems must be addressed:

- tracking service updates and lookups must be fast. Since mobile agents can migrate at any time a huge rate of updates must be expected.

– The load must be distributed between a sufficient number of tracking servers. A suitable unambiguous mapping between agents and tracking servers must be established.
– The number of tracking servers must be gradually scalable to increasing demand.
– tracking services bring security problems that must be addressed properly. Yet heavyweight cryptography (e.g. mutual authentication, public key infrastructures) has an overhead that most probably counters the demand for fast lookups and updates.

In this article we describe our approach to solving these problems. Section 2.1 introduces the notation and conventions we use in this paper. Section 2.2 motivates and describes the architecture and the basic protocols that we developed. *Our principle goal is to reduce application of costly public key cryptography to the bare minimum while achieving a good tradeoff between performance and security.* We also made a reference implementation of these protocols. Section 3 presents the setup and the results of our evaluation of the reference implementation's performance. Related work is discussed in Sect. 4, followed by our brief conclusions in Sect. 5.

2 Tracking Agent Locations

Several models of tracking agents are conceivable. Aridor and Oshima [1] already gave an initial discussion of agent tracking services and suggested three methods of locating agents: *brute force, logging,* and *redirection*. Milojičić et al. distinguish four models [9] which incorporate those of Oshima and Aridor: *updating at home node, registering, searching,* and *forwarding*. We discussed these models in greater depth already in [12], and came to the conclusion that *registering* is our mechanism of choice. *Registering* is a classic server-based approach: one or more dedicated servers provide associative mappings from agent names to agent locations. The interesting part is the structure of the name space, which server is responsible for which part of it, the way updates and lookups are handled, and security mechanisms, of course.

2.1 Notation and Conventions

The description of our protocols uses the notation given below. We will write encryption of some *plaintext* into a *ciphertext* symbolically as $c = \{m\}_K$, where K is the *key* being used. A digital signature will be written as an encryption with a private signing key S^{-1}. We will write $S^{-1}(m)$ when we refer to the bare signature rather than the union of the signature and the signed data. We assume that

the identity of the signer can be extracted from her signature. A cryptographic hash of some input will be written $h(m)$. When A sends some message m to B we will write $A \rightarrow B : m$. For ease of reading, we refer to some entities by their nicknames, e.g. Alice and Nick. In general, Alice plays the role of an agent's owner, and Nick plays the tracking service. For simplicity, we do not distinguish between an entity and its identity, this should become clear from the protocol context. The itinerary of Alice's agent is written as $i_0, \ldots, i_n$, where $i_0 =$ Alice and i_n is the host currently visited by the agent.

2.2 Protocols

We make a number of assumptions on the structure of mobile agents. These assumptions are not required for the tracking service per se, but are important for some of its security properties. Alice prepares her agent Π_ϕ as follows:

$$\Pi_\phi = \{\underbrace{\{\mathcal{P}, \mathcal{S}\}}_{\text{kernel}}_{S_A^{-1}}, \mathcal{V}_0\} \qquad \phi = h(S_A^{-1}(\mathcal{P}, \mathcal{S}))$$

where $\mathcal{P}$ is the agent's program, $\mathcal{S}$ is its static part, and $\mathcal{V}_0$ is its initial variable (or *mutable*) part. Alice signs her agent's program along with its static part for the purpose of authentication and integrity protection. For the sake of security, $\mathcal{S}$ should be unique for each agent instance that uses $\mathcal{P}$ (see [13]). We denote $\{\mathcal{P}, \mathcal{S}\}$ as the agent's *kernel*, and define the agent's name ϕ as a cryptographic one-way hash of the kernel's signature.

We refer to ϕ as the agent's *implicit name*, because it is not a given name but derived from the agent instance itself. The hash function h must be preimage and 2nd preimage resistant (see e.g. [8]). For practical purposes, the SHA (Secure Hash Algorithm) [6] can be used.

Implicit names have a number of useful security properties. Agents cannot impersonate other agents, and the chance to create another agent that maps to the same implicit name (either accidently or on purpose) is negligible. Two agents of the same owner cannot be linked by means of their implicit names. It is virtually impossible to guess agent names. Furthermore, implicit names can be used to bind data, which is acquired by a mobile agent at runtime, securely to that particular agent instance [13].

It may be argued that a random session key could be used rather than an implicit name. However, such a random key would be chosen *explicitly*. This defeats the purpose of the implicit name because a malicious host could provoke a name collision (e.g., it may replace an agent with another of its own that can be controlled conveniently, yet receives all messages directed to the original agent).

Depending on the number of mobile agents that must be served globally, a *scaling factor* l is chosen. The range of hash function h is divided by this scaling factor into 2^l subranges. Each subrange is identified by a number in the range $[0, 2^{l-1}]$. A tracking server is set up for each subrange, and the mapping from subrange numbers to tracking servers is distributed to all agent servers that use the tracking service (see also Fig. 1). As long as few tracking servers are required, this can be done by means of a master list, comparable to the beginning of DNS (Domain Name Service) use. Later on, a special DNS domain can be set up, where the number of the subrange serves as the host name of the tracking server that is responsible for this subrange.

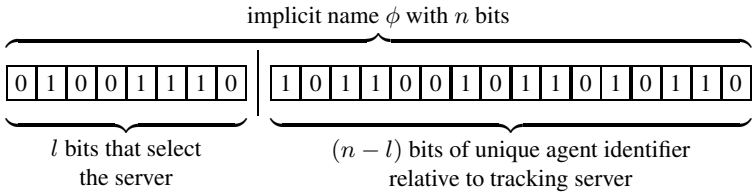

Fig. 1. The implicit name ϕ of length n bits is split into l bits that identify the tracking server, and $n - l$ bits that are used by the tracking server to distinguish between different agents.

What we propose is in fact a global hashtable where each of its slots is managed by a dedicated tracking server. Agents are assigned to slots by means of a cryptographic hash function.

When Alice initializes her agent, she chooses a random initial cookie C_0, and sends her agent on its way with this cookie. The cookie must be big enough to make any chance of being found by guessing or exhaustion attacks negligible. Each hop runs the same protocol when the agent is received:

$$i_{n-1} \rightarrow i_n : \Pi_\phi, C_{n-1} \tag{1.1}$$

$$i_n \rightarrow N : \phi, i_n, C_n, C_{n-1} \tag{1.2}$$

$$N \rightarrow i_n : m \in \{ok, error\} \tag{1.3}$$

$$i_n : \text{executes } \Pi_\phi \text{ until } \Pi_\phi \text{ migrates} \tag{1.4}$$

$$i_n \rightarrow i_{n+1} : \Pi_\phi, C_n \tag{1.5}$$

When i_n receives the agent from i_{n-1} it updates the agent's position in the location register of Nick (the tracking server) with its own name. Nick's name is derived directly from the agent's implicit name and the mapping function, based on the l bits subrange identifier. The update operation is authenticated by

means of the cookie received with the agent. Each host generates and sends a new cookie with its update message. The new cookie is passed together with the agent to the agent's next hop. Hence, each host hands over authority to make location updates to the next hop.

Once an update operation is completed, previous hosts have no access to the location register any more. A host cannot hand off an agent and keep control of the update register at the same time because the next hop would get an error in return of its own update request, thus uncovering the attempt. The host of an agent may nevertheless update the agent's location register with (a series of) bogus locations, thus creating the impression that the agent visited hosts it never visited actually (using IP spoofing if necessary). On the other hand, the host cannot intercept messages that are routed to the agent, without revealing the information sink. We clearly had to make a compromise here, and decided against heavy-weight cryptography in favor of efficiency.

Nick updates its entry for the given implicit name if (1) no mapping of the given implicit name yet exists, or (2) the old cookie C_{n-1} matches the one stored with the location entry. In that case, Nick updates the stored cookie to the new cookie C_n that came with the update request. The location entry is deleted if the host name portion of an authenticated update operation is empty. We refer to this as a *clear* request. Alice sends a *clear* request to Nick when her agent returns, and hosts send clear requests when the agent terminates without migrating.

However, if a tracking server is malicious then it can uncover Alice's identity by observing the host from which either the initial update or the clear request was sent (agents likely return to their owners). Alice can avoid this in two ways: firstly, she never sends the initial update or final clear message – the initial update is then done implicitly by the agent's first hop. Secondly she routes the message through a relay service, for instance another agent of hers, which she sent to a neutral host that allows her agent network access. The important point is that the relay host and the tracking server do not collude.

The *lookup* protocol is also quite simple. Anybody who knows the implicit name of the agent (for instance Alice) can look up its current location by querying the tracking server.

$$A \rightarrow N : \phi \tag{2.1}$$

$$N \rightarrow A : m \in \{i_n, \epsilon\} \tag{2.2}$$

Alice transmits ϕ to a tracking server and receives the current position of Π_ϕ or ϵ if the tracking server does not know ϕ. After a fixed amount of time, Nick expires and garbage collects entries unless they are refreshed. The *refresh* protocol, which is given below, supports bulk refreshes of multiple entries in order to

save bandwidth and to reduce the number of connections that must be opened.

$$i_n \rightarrow N : \phi_1, \phi_2, \ldots, \phi_k \tag{3.1}$$

Please note that, although i_n is given as the origin of refresh requests, they can be sent principally by all entites that know an agent's implicit name. This feature comes handy when attackers attempt to force expiration of a particular entry by launching a *denial of service* attack against the respective agent's current host.

The protocols use few and simple cryptographic operations. In particular public key operations are avoided, which commonly require a public key infrastructure, and carry a heavy burden. On the downside, our protocol does not account for the confidentiality and integrity of the cookies that are transmitted. If Alice is willing to disclose her agent's origin,[1] then she can use a more secure variant of the protocol, whose differences to protocol (1) are the following:

$$A \rightarrow N : \{\phi, i_n, C_n, 0\}_{K_N} \tag{1.0}$$

$$i_n \rightarrow N : \phi, i_n, \{C_n\}_{C_{n-1}}, h(\phi, i_n, C_n, C_{n-1}) \tag{1.2}$$

Alice encrypts the initial update request with the public key of the tracking server. Subsequent updates from agent servers can be protected by means of the chain of cookies, without the use of public key cryptography. This is done by encrypting[2] the new cookie with the old one. The integrity of the data is assured by a message authentication code computed on the implicit name, the new location, and the cookies. Please note that a single leaked cookie allows decryption of all subsequent cookies (given that the adversary intercepts subsequent update requests sent to the tracking server). This is the price that must be paid for not using public key cryptography for all but the first update. Please also note that the current cookie must be transported with the agent. Hence, agents must also be transported over confidential channels for maximum security. Confidentiality is achieved, in the face of network eavesdroppers, only if both conditions are fulfilled.

2.3 Agent Naming and Tracking Framework

The general framework distinguishes between agent *name* services and agent *tracking* services as is illustrated in Fig. 2. This distinction is conceptually comparable to the way file services are implemented in Amoeba [18, §14.6]. Amoeba's *bullet server* serves files based on a flat name space, while the *directory server* maps a hierarchical name space onto the flat name space managed by the bullet server.

[1] Again, Alice can disguise her identity by using a relay agent.

[2] In our reference implementation we use a simple XOR for encryption, which should be sufficient given that the new cookie is generated randomly.

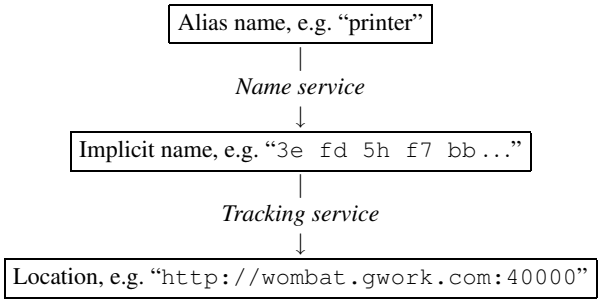

Fig. 2. The framework distinguishes between a name service that maps symbolic agent names onto their implicit names, and a tracking service that maps implicit names onto locations.

We did not yet design the agent name service, but we anticipate that it resolves symbolic, function specific, and user friendly names onto the implicit names required by the tracking service. For instance, a local name service can map the function specific name “printer” onto the implicit name of the nearest printer agent. If a particular printer is to be used then that printer agent’s implicit name can be used to address it unambiguously. The aforementioned distinction also makes sense from a practical and security point of view. Symbolic name mappings are probably more stable than mappings of agent names to agent locations, and they have different security requirements as well, as the given example suggests. However, we have to defer further discussion of the name service, and concentrate henceforth on the design of the tracking service.

Figure 3 gives an overview over the general configuration of the tracking service components. The *tracking server* provides *lookup* and *update* operations as described in Sect. 2.2. Additionally, each tracking server supports a *refresh* operation, and timeouts for its mappings. Periodically, *clients* have to refresh mappings of agents that don’t migrate, in order to keep the mappings valid. Multiple mappings can be refreshed with a single request. The validity period must be well-defined and sufficiently large so that no unreasonable network load incurs. This prevents stale entries from clogging up a tracking server’s database forever.

A *proxy* acts as a write-through cache, and is meant to be an optimization for looking up agents in the administrative domain covered by that proxy, e.g. an organization’s local area network. The use of proxys requires an extension of the protocol: whenever a mobile agent leaves the administrative domain of a proxy, the proxy’s mapping for that agent must be invalidated. Lookups that cannot be resolved by the proxy are forwarded to the appropriate global tracking server.

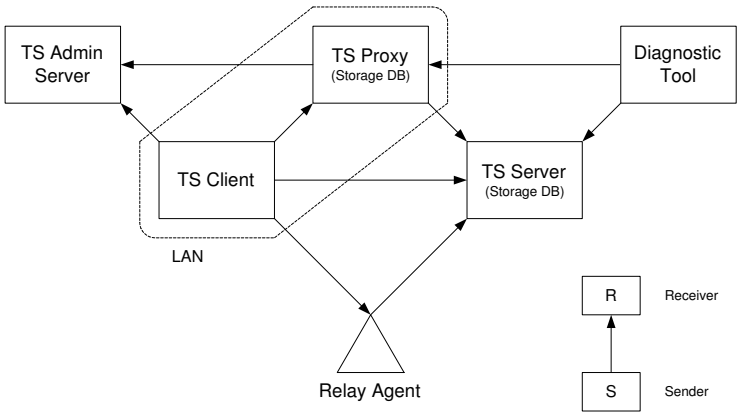

Fig. 3. The agent tracking framework consists of six components: administration servers, tracking servers, proxy servers, clients, relay agents, and diagnostic tools. Arrows in the diagram indicate client/server relationships among these components.

The proxy also assures that lookups for agents in a local area network succeed in the face of a broken link to the external network.

The purpose of the *administration server* is to provide up-to-date information of the scaling factor (the number of bits of each implicit name that identify the subrange of the name space to which the implicit name belongs), and the mapping from subranges to tracking servers. This can be thought of as an alternative to the DNS-based scheme that we mentioned in Sect. 2.2. Since information provided by administration servers is expectantly rather static, regular caching strategies can be used to achieve scalability of this service.

The framework also accounts for a diagnostic component that can be used to query servers and proxys for statistics and excerpts of current mappings (subject to access control). Finally, Fig. 3 shows the relay agent whose relevance for the privacy of agent owners is described in Sect. 2.2.

We made a reference implementation of the tracking server, proxy, client, and a diagnostic tool in the Java programming language. All components were integrated and tested in our mobile agent server SeMoA [14]. The tracking server and proxy run as daemons, which listen on network ports and dispatch requests to a pool of handler threads. Both proxys and tracking servers are backed by a balanced binary search tree [5], which is kept in memory and is accessed by means of a generic interface. Hence, it is straightforward to provide adaptors that interface to a poweful database backend. A detailed description of the reference implementation is beyond the scope of this paper.

2.4 Discussion

Systems such as *Globe* [19] base their scalability on the assumption that some kind of coherence in the movement of mobile objects can be used to optimize the operation of the distributed tracking system. We believe that this assumption is overly optimistic for mobile agents, which operate on a global scale, and whose movement is not bound by physical constraints. With our approach, tracking servers may have to manage agents on the opposite side of the globe (no pun intended). This is the starting point of our discussion below.

Consider $m = 2^l$ name servers. If the probability of a random break down of a name server is equally distributed and Alice would be able to make a random pick among the name servers then her chances of picking one of k broken ones among m are k/m. If she takes our approach then her chances are also k/m given h has a reasonably equal distribution. If Alice knows that a particular name server is down and ϕ is mapped to it then Alice can simply add a random nonce to $\mathcal{S}$ and try again. The chances that she does not succeed after t tries is $(k/m)^t$. If half of the name servers are broken and Alice makes 4 tries then her chance of failing to generate an agent that is mapped to a working name server are 0.0625.

If Alice knows where Π_ϕ is about to go then Alice might want to choose a tracking server with good connectivity to the destinations of Π_ϕ. In our approach, the packets between Bob and Nick might circle the globe in the worst case. However, if Π_ϕ travels around the globe and Alice makes any pick among the name servers then the effects are the same. If Alice still wants to pick one of a particular set of name servers then her chances of succeeding heavily depend on the actual number of name servers and the number of name servers she is willing to take. More precisely, her chances to pick one of k name servers among a total of m servers after t tries is $1 - ((m-k)/m)^t$. If 1% of the name servers are acceptable for Alice then her chances to pick one of those after 4 tries are less than 0.04, and less than 0.1 after 10 tries. Alice will succeed with a probability of approximately 0.9 after 230 tries. On a Pentium II 400MHz we measured less than 4 seconds for computing 230 SHA-1 hashes on about 10K of data (Java implementation).

If a tracking server goes down then entries are certainly lost. However, once the server is available again, agents will be registered as a consequence of regular update request as soon as they migrate. This leaves a window that can be used by malicious servers to "hijack" location entries of agents managed by that tracking server. Nevertheless, we believe that our approach strikes a good compromise between security, scalability, and flexibility.

3 Evaluation Results

Our tests took place in a 100 MBit/s switched LAN that connects a couple of hundreds of workstations and personal computers, and is used by about two hundred researchers and students. We run our software on several Sun Ultra 10 workstations (UltraSPARC-IIi 333 MHz, Solaris 8). The client and proxy machines were equipped with 256 MB main memory, while the tracking server did have 512 MB. We used the HotSpot VM of Java Version 1.3.1 Beta with native thread support and *sunjit* enabled.

First, we tested the capacity and performance of our storage backend. The tracking server was able to hold up to $2 \cdot 10^6$ entries before the system ran out of memory. This means that, given an extreme of $5 \cdot 10^8$ Internet users[3] each running 100 mobile agents simultaneously, about $25,000$ tracking servers would be required to keep all entries. This is less than 0.025% of the hosts in the Internet, according to ISC estimates[4] at the time of writing.

Next, we let up to eight clients send requests concurrently. Table 1 gives the response rates we measured in tests with a single client, sorted by request type. Encrypted registering was slowest, as could be expected. However, this type of request is required only once per agent. In this test the tracking server handled about 200 agent lookup requests per second, which includes processing overhead at the client (clients start requests sequentially). Figure 4 shows the response rates we measured for concurrent lookup requests with one to eight clients. With two or more clients, the response rate jumps from about 200 requests per second to roughly 325, and remains more or less stable at this mark (with one client, the server has idle time, with two or more it becomes congested). Table 4 shows how response times develop with an increasing number of clients. With about 80 clients, requests take longer than 15 seconds to process, which causes network connections to time out.

We also measured the impact of the tracking service integration on the migration time of mobile agents in the SeMoA server. Without tracking service integration, we measured an average of 1.178 seconds per migration of a simple benchmark agent, compared to 1.18 with location tracking, which we consider tolerable.

[3] NUA estimates there were more than 400 million users online in the Internet on December 2000, source: `http://www.nua.com/surveys/how_many_online`

[4] ISC estimates there were more than 100 million hosts in the Internet on January 2001, source: `http://www.isc.org/ds`

Type	Length	Mean time	Requests/s	Comment
lookup	30 bytes	4.7 ms	213	tracking
register encrypted	421 bytes†	201 ms	5	init
update	103 bytes†	8 ms	125	update
register plain	103 bytes†	5 ms	200	proxy

Table 1. This figure shows the size of request packets, and average processing time of the tracking service with one client, by request type. The lengths marked with † may differ depending on the length of the stored location reference.

4 Related Work

The *Globe* [19] system is a distributed directory designed to support billions of references to mobile objects. However, the authors acknowledge that their hierarchical approach is not scalable enough to fulfill this goal due to the enormous storage demands and relatively large number of requests that must be handled by higher-level directory nodes. In order to overcome these problems, they propose to use the first n bits of an object's globally unique handle as the identifier of directory *subnodes*, which share the load on their directoy level. This approach equals the one we chose in order to provide scalability.

The notion of *put-ports* and *get-ports* in *Amoeba* [18, pp. 607] can be regarded as an analogy to the *implicit* naming scheme we propose for agents in this article. An *Amoeba* server process registers with the *Amoeba* kernel using a (private) *get-port*, and the according *put-port* is computed by the kernel by means of a one-way function. Processes that wish to communicate with the server process address packets to the *put-port*. This prevents intruders from impersonating server processes. The server process can be regarded as a mobile agent, and the *put-port* as its name. In contrast to *Amoeba* we do not allow free choice of the private *get-port*, but compute the *put-port* directly from the agent's unique kernel, which is required to prevent one agent from impersonating another with the help of a malicious host that leaks the equivalent of *get-ports*.

Several other schemes for locating mobile agents, and routing messages among them were proposed in the past, e.g. [3, 11, 4, 20, 17, 7]. Some of these approaches assume that there is a logical network of connected agent servers [3, 11, 7], and routing of agents or messages is done along the edges of this graph. In the case of [7], the graph must actually be a balanced tree. However, any approach that builds on a particular network topology makes sense only if mobile agent systems are implemented on the network layer as part of routers. Most of the contemporary mobile agent systems are implemented on the application layer, though. From the perspective of the application layer, the Internet is a fully connected graph. Hence, a logical topology that is layered on top of

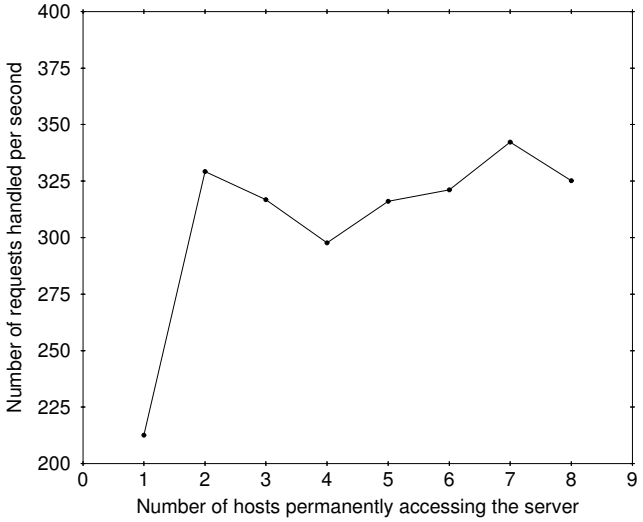

Fig. 4. This figure shows the average number of requests that can be handled by the tracking server, depending on the number of clients that query the server concurrently. A circle mark represents the mean of a set of 4500 measured values.

the physical structure of the Internet creates undesired and unnecessary routing overhead. The logical routing may even run counter to the actual physical routing. Additionally, the approaches described in [3, 7] put the burden of setting up and maintaining the logical structure on administrators; a job that, in our opinion, quickly spirals out of control.

In particular, the approach described in [3] is not scalable. *Each* node in the tree has storage requirements proportional to the number of mobile agents managed by it, and update rates proportional to the rate of migrations that start or end in its subtree. In particular, the root node has to cope with *all* of the traffic.

Strategies based on forwarding pointers and dynamic forshortening of pointer chains are proposed, e.g. in [16, 10]; they are also used in *Mole* for the purpose of orphan detection [2]. The disadvantage of this approach is its lack of robustness, a single broken or timed-out link makes the agent unreachable.

The *Mobile Object Workbench* [4] supports a hierarchical directory service for locating objects that moved. Wojciechowski *et al.* [20] use a combination of registering and forward references. Forward references act as a cache. In case of a miss, the central server is asked to forward the message, and the invalid forward reference is updated. Di Stefano *et al.* [17] propose the use of location

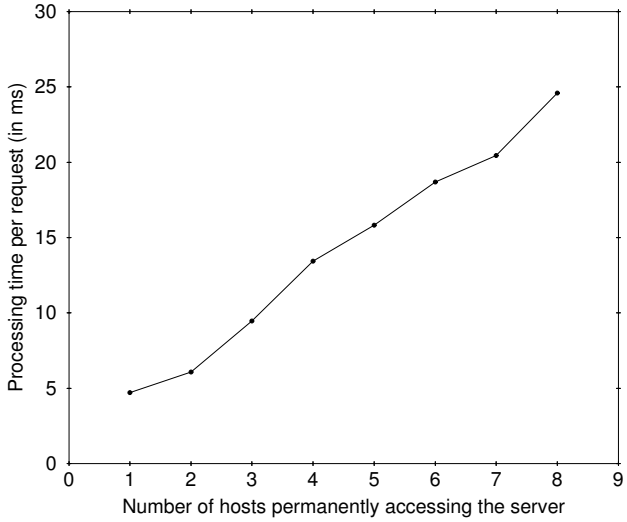

Fig. 5. This figure shows how response times of the tracking server develop with an increasing number of clients that queries the tracking server concurrently. A circle mark represents the mean of a set of 4500 measured values.

servers, where each server is responsible for all agents in its domain. Each agent has a home server that can be derived from a location-specific part of the agent's name. Whenever the agent enters a new domain, the servers responsible for the old and new domain, as well as the home server are updated. Lookups for agents not in the local domain start at the home server.

A detailed discussion of all these approaches is beyond the scope of this paper, and is well worth a paper of its own. To the best of our knowledge, none of the approaches described above address security issues, and few seriously address Internet-wide scalability.

5 Conclusions

In our paper, we propose a framework and protocols for a secure and scalable global tracking service for mobile agents. We do not presuppose that a mobile agent's migration pattern is governed by a coherence principle that could be used to achieve scalability. We must anticipate a high rate of update requests of agent locations, and thus our approach was designed to scale without caching

mechanisms. Our approach resembles a global hash table, where the hash function fulfills some security requirements.

The protocols we devise have a number of advantageous security properties. In particular, malicious location updates by unauthorized hosts are prevented. Each host must hand off authority to update an agent's location register when the agent migrates. Agents cannot impersonate other agents with regard to their names. Furthermore, the implicit naming scheme for agents prevents malicious tracking servers from profiling agent owners by means of their agents' movements, given that an agent's host does not collude with the adversary. This protects the privacy of agent owners in the face of omnipresent tracking servers. All this is achieved with a minimum of cryptographic overhead, which is an important requirement for scalability.

Furthermore, we made a reference implementation of our protocols, and present the results of its evaluation. The outcoming is as good as one could expect from a Java implementation, and can be further improved by implementations that are optimized for the processor architecture of the machines on which the tracking server shall run. We are well aware that laboratory settings hardly give significant evidence for a system's applicability when used in the field. Therefore, we would very much like to tests our implementation in a larger scale, and welcome interested parties that would like to take part.

6 Acknowledgements

This paper elaborates on initial ideas presented in [12]. In particular, it contributes to the proxy approach, the encrypted update protocol, as well as a report on the evaluation of our reference implementation. We'd like to thank the anonymous reviewers for their detailed and constructive comments which helped us to improve the paper.

References

1. Yariv Aridor and Mitsuru Oshima. Infrastructure for mobile agents: Requirements and design. In Rothermel and Hohl [15], pages 38–49.
2. Joachim Baumann and Kurt Rothermel. The Shadow Approach: An orphan detection protocol for mobile agents. In Rothermel and Hohl [15], pages 2–13.
3. L. Bernardo and P. Pinto. A scalable location service with fast update responses. In *IEEE Global Telecommunications Conference (GLOBECOM '98)*, volume 5, pages 2876–2881, 1998.
4. Michael Bursell, Richard Hayton, Douglas Donaldson, and Andrew Herbert. A Mobile Object Workbench. In Rothermel and Hohl [15], pages 136–147.
5. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1996.

6. FIPS180–1. Secure Hash Standard. Federal Information Processing Standards Publication 180–1, U.S. Department of Commerce/National Bureau of Standards, National Technical Information Service, Springfield, Virginia, April 1995. supersedes FIPS 180:1993.
7. S. Lazar, I. Weerakoon, and D. Sidhu. A scalable location tracking and message delivery scheme for mobile agents. In *Proceedings Seventh IEEE International Workshop on Enabling Technologies: Infrastucture for Collaborative Enterprises (WET ICE '98)*, pages 243–248. IEEE Computer Society Press, 1998.
8. Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. Discrete Mathematics and its Applications. CRC Press, New York, 1996. ISBN 0-8493-8523-7.
9. D. Milojicic, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *Proc. of the Fourth USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)*, April 1998.
10. Luc Moreau. Distributed directory service and message routing for mobile agents. Technical Report ECSTR M99/3, Department of Electronics and Computer Science, University of Southampton, November 1999.
11. Amy L. Murphy and Gian Pietro Picco. Reliable communication for highly mobile agents. In *Proc. First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents (ASA/MA '99)*, pages 141–150. IEEE Computer Society Press, 1999.
12. Volker Roth. Scalable and secure global name services for mobile agents. 6th ECOOP Workshop on Mobile Object Systems: Operating System Support, Security and Programming Languages (Cannes, France, June 2000).
13. Volker Roth. Programming Satan's agents. In *Proc 1st International Workshop on Secure Mobile Multi-Agent Systems*, Montreal, Canada, May 2001. Available at URL `http://www.dfki.de/~kuf/semas/semas-2001/`.
14. Volker Roth and Mehrdad Jalali. Concepts and architecture of a security-centric mobile agent server. In *Proc. Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)*, pages 435–442, Dallas, Texas, U.S.A., March 2001. IEEE Computer Society. ISBN 0-7695-1065-5.
15. K. Rothermel and F. Hohl, editors. *Proceedings of the Second International Workshop on Mobile Agents (MA '98)*, volume 1477 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin Heidelberg, September 1998.
16. Peter Sewell, Pawel Wojciechowski, and Benjamin Pierce. Location-independent communication for mobile agents: a two-level architecture. Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.
17. A. Di Stefano, L. Lo Bello, and C. Santoro. Naming and locating mobile agents in an Internet environment. In *Proc. Third International Enterprise Distributed Object Computing Conference (EDOC '99)*, pages 153–161, 1999.
18. Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall, Inc., 1992.
19. M. van Steen, F. J. Hauck, P. Homburg, and A. S. Tanenbaum. Locating Objects in Wide-Area Systems. *IEEE Communications Magazine*, pages 104–109, January 1998.
20. Pawel Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. In *Proc. First International Symposium on Agent Systems and Applications, and Third International Symposium on Mobile Agents (ASA/MA '99)*, volume 2, pages 821–826, October 1999.