SECMAP: A Secure Mobile Agent Platform

Suat Ugurlu and Nadia Erdogan

Istanbul Technical University, Computer Engineering Department,
Ayazaga, 34390, Istanbul, Turkey
suat@suatugurlu.com, erdogan@cs.itu.edu.tr

Abstract. This paper describes a mobile agent platform, Secure Mobile Agent Platform (SECMAP), and its security infrastructure. Unlike other agent systems, SECMAP proposes a new agent model, *the shielded agent model*, to meet security requirements and provides functionalities which ensure the implementation of the the shielded agent model. It provides secure agent communication and migration facilities, and maintains security policy information to examine agent actions and to prevent undesired/unauthorized activity, while employing cryptographic techniques to meet security constraints.

1 Introduction

A mobile agent is a program that has the autonomy to travel around a network to accomplish its tasks [1][2]. Mobility involves the movement of executable code and associated execution state between different hosts on the network. A mobile agent is executed in an environment called a mobile agent platform which is a distributed abstraction layer that provides mechanisms for both communication and mobility support.

Any piece of code which is run on a computer system can potentially threaten the security, privacy, and integrity of the system and its users [3]. Security issues have gained new importance with the extensive use of mobile code systems. Any mobile code platform suffers from four basic categories of potential security threats[4]:

Leakage: unauthorized attempts to obtain information belonging to or intended for someone else
Tampering: unauthorized changing (including deleting) of information
Resource stealing: unauthorized use of resources (e.g., memory, disk space)
Antagonism: interactions not resulting in a gain for the intruder but annoying for the attacked party.

Meeting security requirements is fundamental to mobile agent systems and an inability to provide a feasible agent security model seriously hinders a wider adoption of mobile code based applications. An acceptible mobile agent based system requires secure techniques for agent migration and communication, and also mechanisms for higher level security management and maintenance.

This paper describes a new mobile agent platform, SECMAP, that especially focuses on security issues present in agent systems. Unlike other agent systems,

M. Pěchouček, P. Petta, and L.Z. Varga (Eds.): CEEMAS 2005, pp. 102 – 111, 2005.

SECMAP proposes a new agent model, the *shielded agent model*, for security purposes. A shielded agent is a highly encapsulated software component that ensures complete isolation against unauthorized access of any type. SECMAP provides secure agent communication and migration facilities, and maintains security policy information to examine agent actions and to prevent undesired/unauthorized activity. Additionally, SECMAP continuously monitors and reports on the execution of an agent from its creation to its completion.

2 Security Model of SECMAP

In a mobile agent system, agents cannot be reliably associated with end users without taking certain precautions. The approach taken by SECMAP is to treat every agent as a distinct principal and to provide protection mechanisms that isolate agents. SECMAP differs from other mobile agents systems in the abstractions it provides to address issues of agent isolation. SECMAP provides a light-weight implementation of agents; they are implemented as threads instead of processes. Each agent is an autonomous object with a unique name.

A Secure Mobile Agent Server (SMAS) resident on each node presents a secure execution environment on which new agents may be created or to which agents may be dispatched. SMAS provides functionalities that meet security requirements and allow the implementation of the *shielded agent model.* A shielded agent is a highly encapsulated software component that ensures complete isolation against unauthorized access of any type. On a request to create a new agent, SMAS instantiates a *private object* of its own, an instance of predefined object *AgentShield*, and uses it as a wrapper around the newly created agent by declaring the agent to be a *private object* of *AgentShield* object. This type of encapsulation ensures complete isolation, preventing other agents to access the agent state directly. An agent is only allowed to communicate with its environment over the SMAS engine through the methods defined in a predefined interface object, *AgentInterface*, which is made the private object of the agent during the creation process. The interface provides limited yet sufficient functions for the agent to communicate with SMAS. All variables of agents are declared as private and they have corresponding accessor methods.

SECMAP allows the concurrent execution of several agents on the same host and each agent runs as a separate thread in the same memory area of the host. In this mode of operation, the shielded agent model suffices to guarantee inter agent isolation and protection. Figure 1 depicts the layered structure of a shielded agent. SECMAP employs cryptographic techniques to meet security constraints. Each SMAS owns a certificate which is used to identify its identity and to encrypt and decrypt data. A requests from a SMAS is not processed before the validity of the SMAS identitiy is verified. A SECMAP agent's code and state information are kept encrypted during its life time using Data Encryption Standard (DES) algorithm. They are decrypted only when the agent is in *running state* on the host's memory. Thus, an agent is identified as a black box on a host, except while in memory. To protect agents during migration over the network, agent code and state data are encrypted as well while in transfer and can only be decrypted on the target host after retrieving the appropriate DES key from the security manager.

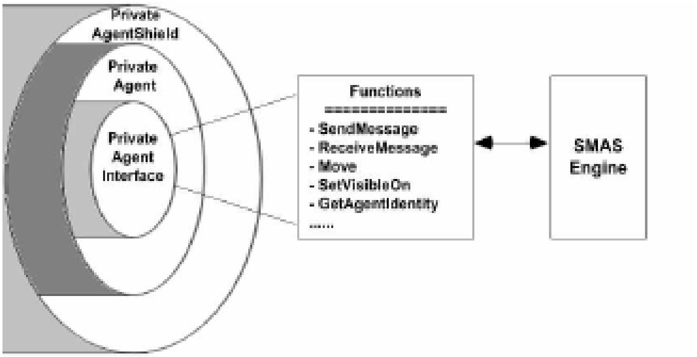

Fig.1. The shielded agent model

SECMAP employs a policy based authorization mechanism to permit or restrict agents to carry out certain classes of actions. Agent communication, migration, disk I/O, access to system resources are some of the events that require enforcement of security policies. SECMAP allows for policies to be dynamically defined and be enforced by intercepting agent service requests. It monitors, time stamps and logs all agent activity in a file, in order to be later analyzed to determine the actions an agent carried out on the host. In case an unexpected result is recognized, the route of the agent can be traced and how the agent was executed on each host can be found. In addition, in case of a threat, SMAS has the privilege hinder the activites of an agent. This is accomplished by purging all agent-related variables known to the SMAS such as Agent policy, Agent Identity, Agent message queues, etc. Under this condition, the agent can no longer be effective as any attempt to communicate or to carry out actions monitored by the security manager will lead to exceptions. However, if the agent includes a piece of code such as "*while (true) { }*", it will continue its execution and consume CPU time. SECMAP can not prevent this kind of an attack.

3 SECMAP Architecture

Figure 2 shows the SECMAP architecture. The main component of the architecture is a Secure Mobile Agent Server (SMAS) which provides a platform in which agents exist and interact with other agents. In order to execute agents, each computer node must host a SMAS. SMAS is responsible of agent management tasks such as creation or activation, agent communication, agent migration, and policy management, each contributing to the implementation of the *shielded agent* model. Furthermore, having full control over agent activities, SMAS can identify what an agent attempts to do and if it has the rights. A second component of the architecture is an API which works as the interface between an agent and its SMAS platform. An agent can request communication or migration via this interface. The interface has limited functions and is the only way for the agent to interact with its environment.

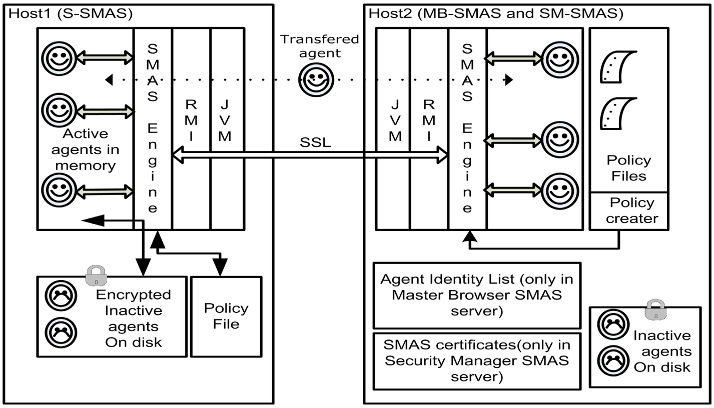
Host1 (S-SMAS)
SMAS Engine
RMI
JVM
Active agents in memory
Transfered agent
SSL
Host2 (MB-SMAS and SM-SMAS)
JVM
RMI
SMAS Engine
Policy Files
Policy creater
Encrypted Inactive agents On disk
Policy File
Agent Identity List (only in Master Browser SMAS server)
SMAS certificates(only in Security Manager SMAS server)
Inactive agents On disk

Fig. 2. SECMAP Architecture

3.1 SMAS Modes of Operation

A SMAS may operate in three modes according to the functionality it exhibits. It can be configured to execute in any of the three modes on a host through a user interface.

Standard Mode (S-SMAS): S-SMAS provides standard agent services such as agent creation, activation, inactivation, removal, communication, and migration. It also includes a policy engine that checks agent activity and resource utilization according to the rules that are present in a policy file, which has been received from a Security Manager SMAS. In addition, S-SMAS maintains a list of all active agents resident on the host and notifies the Master Browser SMAS anytime an agent changes state. Keeping logs of all agent activities is another important task S-SMAS carries out. Log content may be useful in the detection of attacks which are difficult to catch instantly.

Master Browser Mode (MB-SMAS): When agents are mobile, location mappings change over time, therefore agent communication first requires a reference to the recipient agent to be obtained. In addition to supporting all functionalities of S-SMAS, MB-SMAS also maintains a name-location directory of all currently active agents in the system. This list consists of information that identifes the host where an agent runs and is kept up to date as information on the identities and status (active/inactive) of agents from other SMAS is received.

Security Manager Mode (SM-SMAS): In addition to supporting all functionalities of S-SMAS, SM-SMAS performs authentication of all SMAS engines, handles policy management, and maintains security information such as DES keys and certificates.

Every SMAS engine has a module to create its self signed certificate. The private key that the SMAS has created for itself is kept in its secure place and the public key is sent to the SM-SMAS. The programmer managing the SM-SMAS can import this public key into the key store of SM-SMAS so that SM-SMAS can trust the SMAS engine. SMAS also should import the SM-SMAS public key into its key store as well to recognize the SM-SMAS as a trusted communication party. No SMAS engine whose public key is not imported into the SM-SMAS key store can communicate with the SM-SMAS since this is also the requirement of SSL which is used as the communication protocol under RMI in SECMAP. Since agents of different SMAS will need to communicate with each other, a SMAS engine can request the certificates of all other SMAS engines from the SM-SMAS and import them into its key store in order to be able to recognize them as trusted parties. Up to this point, all requirements for encrypted communication are provided but still there is a strong need to distinguish who is who. SM-SMAS also creates authentication keys for each of SMAS in the system. After establishing a SSL session, any SMAS should be authenticated by the SM-SMAS before it can start up as a trusted server. SM-SMAS holds an IP address and key pair for each of SMAS engine that wants to be authenticated. If the supplied key and the IP address of the requesting SMAS engine are correct then it is authenticated. Once authenticated, SM-SMAS recognizes the SMAS. Every authenticated SMAS engine gets a ticket with a specified life time from the SM-SMAS, and uses this ticket whenever it attempts to start communication with other SMAS engines. The target SMAS engine first refers to SM-SMAS to verify the validity of the ticket before proceeding with the necessary actions to fulfill the communication request. This mode of operation prevents any untrusted entity in the network to masquerade as a valid SMAS.

The system is managed with a decentralized control; several MB-SMAS and SM-SMAS may be active and cooperate for a smooth execution. They share their data and keep it coherent. When initializing a S-SMAS on a node, the programmer specifies the addresses of the MB-SMAS and the SM-SMAS it should register to. S-SMAS sends its agent list to MB-SMAS and, in return, receives the identities of all other agents active on the system. We call those S-SMAS that a MB-SMAS or a SM-SMAS cooperates with as its *partners*. When a MB-SMAS gets a request to return an agent identity, it cooperates with its partners to obtain the current agent identities. A similar mode of processing is true for SM-SMAS. If a SM-SMAS can not authenticate the request, directs it to its partners for possible authentication. Additionally, when a S-SMAS communicates its MB and SM-SMAS, it obtains the addresses of their partners and saves them, in order to use as a contact address in case its communication to its MB-SMAS or SM-SMAS fails. This approach adds robustness against network or node failures.

An important component of SM-SMAS is the policy creator. Policy creator can create different sets of policies and install them on different SMAS engines.

Agent activities related to resource usage such as disk I/O or creation of network connections directly by using socket objects are first checked by the SMAS engine and blocked if not coherent with its security policy. SMAS engine achieves this by creating a *custom java security manager* monitoring all resource accesses. SMAS also enforces security policies to permit or restrict other agent activities such as messaging and migration.

3.2 Security Policies

SECMAP guarantees that an agent performs only the activities that it is permitted by verifying each action request by the agent against a set of policy rules. Security policies are created by SM-SMAS and sent to other SMAS. Policy rules can be defined for the following purposes:

- An agent can be restricted to communicate with only certain agents, with only agents on a certain SMAS or can be totally restricted to send and receive messages. Restrictions can be applied to sending or receiving separately.
- An agent may be restricted to migrate to only certain hosts or a host may be restricted to not accepting any agent from certain other hosts.
- An agent may be restricted to not performing disk I/O on the host it is running on or only specific agents may be allowed to carry out specific disk operations.
- An agent can be restricted to not creating or accepting socket based connections to other applications on the network. A host's socket factory may be totally prohibited to be used by any agent.
- An agent's access to system variables of the host may be restricted .

Imposing time-based restrictions for all types of rules is also possible. Furthermore, there are other security settings that are configured on SM-SMAS, however not in the form of a security policy. For example, an agent's size can be restricted to an upper limit in order to prevent an agent to use a host's memory and cause a memory leak.

The use of policies results in a more dynamic execution environment. Restrictions on agent activities may be altered at any point in time during execution with an appropriate modification of the agent policy, requiring no change in the agent code. With this approach, a higher level of security and also of flexibility is attained.

4 SECMAP Agents

SECMAP requires agents to conform to a software architectural style, which is identified by a basic agent template shown in Figure. 3. The agent programmer is provided a flexible development environment with an interface for writing mobile agent applications. He determines agent behavior according to the agent template given and is expected to write code that reflects the agent's behavior for each of the public methods. For example, code for the *OnCreate()* method should specify initial actions to be carried out, or code for the *OnMessageArrive()* method should define agent reaction to message arrival. In accordance with this style, an agent may be in one of different states throughout its existence and exhibits the following behavior:

```
public class Main extends Agent{
  public void OnMessageArrive(){... }
  public void OnCreate(){ ... }
  public void OnActivate(){... }
  public void OnInactivate(){... }
  public void OnTransfer(){... }
  public void OnEnd(){... }}
```

Fig. 3. Agent Template

*State on_create***:** On an initial creation, a unique identity, an instance of class `AgentIdentity` is defined for the agent. An agent is referenced through its identity, which consists of three parts. The first part, a random string of 128 bytes length, is unique identification number and, once assigned, never changes throughout the life time of the agent. The second part is the name which the agent has announced and wishes to be recognized with. While the first two parts are static, the third part of the identity has a dynamic nature: it carries location information, that is, the address of the SMAS on which the agent is currently resident, and varies as the agent moves among different nodes. This approach facilitates efficient message passing.

State on_activate: An agent becomes active and starts executing while in this state. An agent should be active in order to be able to communicate with other agents. A programmer may prefer not to specify any code for this state, if just activating the agent meets his goals. He can then program the *OnMessageArrive* method of the agent to send and receive messages.

State on_inactivate: When an agent enters this state, its execution is stopped and its context (data, variables and code) is saved in the SMAS agent directory. The agent can not send or receive messages while in this state.

State on_transfer: An agent may request to migrate to another host anytime while it is active. SMAS inactivates the agent before the transfer begins, interacts with the remote SMAS to transfer its code and state data, and if the transfer operation completes successfully, deletes the agent from the local SMAS agent directory. Meanwhile, the remote SMAS re-creates the agent in its last state and activates it so that it starts execution.

State on_end: The agent is removed from the local SMAS on successful migration. The agent template prevents agents from sharing information through static variables, consequently eliminating the possibility of backdoor communication.

4.1 Agent Communication

SECMAP agents communicate via messages. SMAS supports asynchronous message exchange primitives through methods of *AgentInterface*. Agent communication is secured by transferring encrypted message content through SSL. Agents are provided with a flexible communication environment where they can question the results of message send requests, wait for responses for a specified period of time, and receive messages or replies when it is convenient for them. Figure 4 shows the communication framework and how a request to send a message proceeds. During agent creation, SMAS, while instantiating a shield object for the agent, also creates three queues: one for outgoing messages, one for incoming messages and one for reply messages. The input and output queues are monitored by two threads which are spawned on agent activation. The thread monitoring the input queue alerts the agent if a message arrives, while the thread monitoring the output queue alerts the SMAS engine to route messages to their destination.

Communication is asynchronous. When an agent issues a send message call through the *AgentInterface*, the message is placed into the output queue by the agent shield and the call returns. From then on, the agent may continue with its operations. It may question the result of the send request, or, if it expects a response, it may retrieve the reply message at any point suitable in its execution path. The thread monitoring the output queue alerts the SMAS engine to route the message. After the SMAS on the recipient host places the message into the input queue of the target

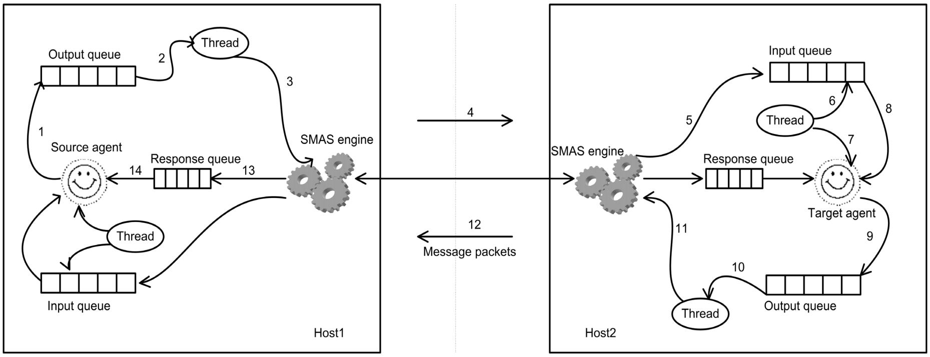

Fig. 4. Agent Communication Framework

agent, the input queue thread alerts the agent of the arrival of a new message. Subsequent to being alerted, the agent can issue a call to receive the message at any time. Reply messages are also routed as regular messages are. The only difference is that the SMAS engine sending the reply sets the acknowledgement field at the end of the message packet object so that the message can be placed in the reply queue of the agent to which the call returns a result. The reply can be retrieved at any time.

Before an agent can send a message to another agent, it needs to learn the name of the receiver agent. An agent learns the identity of the target agent via a call to the SMAS, which cooperates with MB-SMAS to return the required information, an object of type is `AgentIdentity`. Messages are created as instances of the *Message* class and consist of two parts. The first part is the name of the message, while the second consists of a parameters. Parameters can be of any type that can be serialized.

4.2 Agent Migration

SECMAP supports weak migration of agents between remote hosts on a call to the *Move* method of *AgentInterface*. The agent issues a *Move(address)* call to migrate to another host. The call returns a result object through which the agent can question the result of the transfer request. The *address* field of the call specifies the remote SMAS, IP address of the host remote SMAS is running on and the name of the remote SMAS The SMAS engines involved in the migration process carry out the following steps:

– The agent is inactivated on local SMAS and an inactivation information message is sent to MB-SMAS.

– Agent code and state information are saved in the local SMAS agent directory. All class files belonging to the agent are zipped into a single file in order to reduce agent transfer time. Agent state is written into another file. Once agent code and state is written into the disk, they are encrypted and no one can decrypt them since only SM-SMAS has the correct key. Local SMAS gets the key from the SM-SMAS.

– The agent code and state are transferred to the remote SMAS. The remote SMAS re-creates the agent, loads its code, state and identity from the transferred files after decrypting them and activates the agent. As agent code and state are held in an encrypted and zipped file, a customized class loader rather than the system class loader is used. The loadClass() method of the newly developed AgentClassLoader has been enhanced with new capabilities in order to complete this phase of migration. Once the agent is activated, an acknowledgement message of activation information is sent to MB-SMAS so that it can update the agent's location information in order to be able to redirect any new message destined to this agent to the correct SMAS.

– The agent is deleted from the source SMAS if the transfer is successful. If any of the steps described above fails, SMAS cancels the transfer.

5 Related Work

Developers and researchers have taken a variety of approaches to security of mobile agent environments. Hohl [5] proposes what he refers to as Blackbox security to scramble an agent's code in such a way that no one is able to gain a complete understanding of its function. Proof carrying code [6] requires the author of an agent to formally prove that the agent conforms to a certain security policy. By digitally signing an agent, its authenticity, origin, and integrity can be verified by the recipient. The idea behind path histories [7] is to let a host know where a mobile agent has been executed previously. State appraisal [8] attempts to ensure that an agent's state has not been tampered with through a state appraisal function which becomes part of the agent code. In general, there does not seem to be a single solution to the security problems introduced and most of the solutions are inadequate in protecting agent and host data, while others that provide adequate protection cause an unacceptable overhead to the programmer. However, work is still going on, and new system are being developed [9] [10].

6 Conclusions and Future Work

This paper describes a mobile agent platform, SECMAP, and its security infrastructure. The system has been especially developed against security threats that both agents and hosts may be exposed to. Security features are inserted into the system core at design time. The system has an open and flexible architecture that can further be enhanced in the future to meet additional requirements.

SECMAP allows for completely isolated lightweight agents through a new shielded agent model which protects the agent from its environment, while, at the same time, providing secure, flexible and efficient communication facilities. SECMAP introduces trusted nodes into the infrastructure, to which mobile agents can migrate when required, so that sensitive information can be prevented from being sent to untrusted hosts. Sources of requests are authenticated before they are processed to verify that they really come from their stated, trusted sources. This approach does not appear to be fully explored elsewhere. The built in support to secure agent communication and migration relieves the programmer of extra coding, providing a transparent execution environment. SECMAP employs cryptographic techniques to meet security constraints. An agent's code and state information are kept encrypted during its life time, being decrypted only when the agent is in *running state* on the host memory. Thus, an agent is identified as a black box on a host, except while in memory. Unlike several agent systems, SECMAP employs policy rules to protect not only hosts but also agents. An agent's capabilities such as, communication, migration, I/O, socket communication can be totally or partially restricted. Policies can be changed after agent deployment as well. SECMAP monitors and records all agent activities. These traces can be used not only for debugging purposes but also for security purposes. An intelligent analysis of these records may provide additional security benefits and can help to detect certain kinds of attacks which are normally very difficult to detect.

Currently, work is in progress on detection and resolution of policy conflicts and enforcement of security policies. Our future work also includes the addition of dynamic policy creation capability to the architecture with the help of log analysis.

References

1. S. Franklin and A. Graesser "Is it an Agent, or just a program? A taxonomy for Autonous Agents" Proc. Third International Workshop on Agent Theories, Architecures,and Languages, Springer -Verlag,1996.
2. Karnik, N.M., Tripathi, A.R., 1998. Design issues in mobile-agent programming systems. IEEE Concurrency 6 (3), 52–61.
3. M. Hauswirth, C. Kerer, and R. Kurmanowytsch, "A flexible and extensible security framework for Java code", Technical Report TUV-1841-99-14, Technical Univ. of Vienna.
4. G. Coulouris, J. Dollimore, and T. Kindberg. Security. In *Distributed systems - concepts and design*, International Computer Science Series, pages 477-516, 2nd edition. Addison-Wesley, Reading, Mass. and London, 1994.
5. F. Hohl. "Protecting mobile agents with blackbox security" Proc. 1997 Wksp. Mobile Agents and Security , Univ. of Maryland , Oct 1997
6. G. C. Necula and P. Lee. "Safe, untrusted agents using proofcarrying Code" In Giovanni Vigna, editor, Mobile Agents and Security, Number 1419 in LNCS, pages 61-91. Springer-Verlag, Berlin, 1998.
7. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik, "Itinerant agents for mobile computing", In M. N. Huhns and M. P. Singh, editors, Readings in Agents, pages 267-282. Morgan Kaufmann, San Francisco, CA, 1997.
8. W. Farmer, J. Guttmann, and V. Swarup, "Security for mobile agents: Authentication and state appraisal", In E. Bertino, H. Kurth, G. Martella, and E. Montolivo, editors, Proc. of ESORICS 96, Number 1146 in LNCS, pages 118-130. Springer-Verlag, Berlin, 1996
9. C. Bryce, J.Vitek, "The JavaSeal Mobile Agent Kernel", Autonomous Agents and Multi-Agent Systems, 4, 359-384,2001
10. V.Varadharan and D.Foster, "A Security Architecture for Mobile Agent Based Applications" World Wide Web:Internet and Web Information System, 6,93-122, 2003