AGENT MONITORING AGENTS & DEVELOPMENT OF A GROUP CLASS

Marietta A. Gittens, Shu Shen
Sadanand Srivastava, James Gil Delamadrid
{mgittens,sshen,ssrivas,gildelam}@cs.bowiestate.edu
Department Of Computer Science,
Center for Distributed Computing (CeRDiC)
Bowie State University, MD, 20715.

This paper deals with the development of a **group** class. This abstract class inherits from our previously developed **Agent** superclass. All instances of groups inherit from the **ComWin** class, which is an extension of the **group** class, and which accepts, via its command window, user commands of specific actions to be performed on agents or groups of agents. Each instance of a group becomes a group leader that possesses the capability of **accepting** as many members as possible, **deleting** members from its association, **communicating** messages to one or all of its members, **destroying** itself or any or all of its members, accepting leadership of a group that is **merging** with it, returning leadership to the former leader of a subgroup by **splitting** that group away and **moving** its members to a new host to continue its operations there.

Such a **group** class was developed to extend the capabilities of our Agent Monitoring System. At present our system is capable of monitoring the occurrence of communication exchanges between agents and detecting whether agents are active or not. With the incorporation of our **group** class to the system, we hope to extend the Agent Monitor s capabilities to include monitoring the timestamp and location of the above mentioned events, the reason for their occurrence, and the identity of the participating group leaders or agents.

The group class offers the user greater flexibility and control over a wide collection of agents performing a variety of specified tasks on his/her behalf. If the user no longer requires the services of a particular agent, he can remotely order that agent to **destroy** itself. If, however, some new service is desired, he can **communicate** to the agent accordingly. A visual representation of group members and the hierarchical relationships among group instances are displayed on our user interface via the newly developed **gpCanvas**.

AMA Architecture: Our AMA System consists of two main parts: an Agent Monitoring Agent (AMA) and an agent-based system. The AMA consists of the **AgentMonitor** (our server) and its associated user interface, **amaPage,** which, via its three canvases, show agent activity, group hierarchies and agent communication. The agent-based system consists of instances of communicating agents which may or may not belong to groups (Fig.1)

Fig.1 THE AMA STRUCTURE

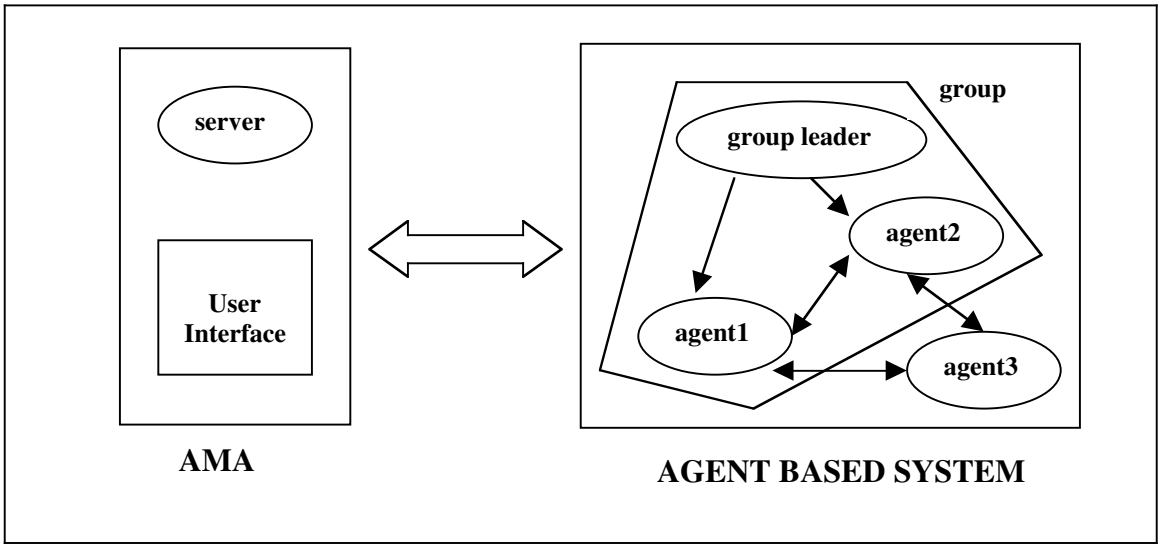
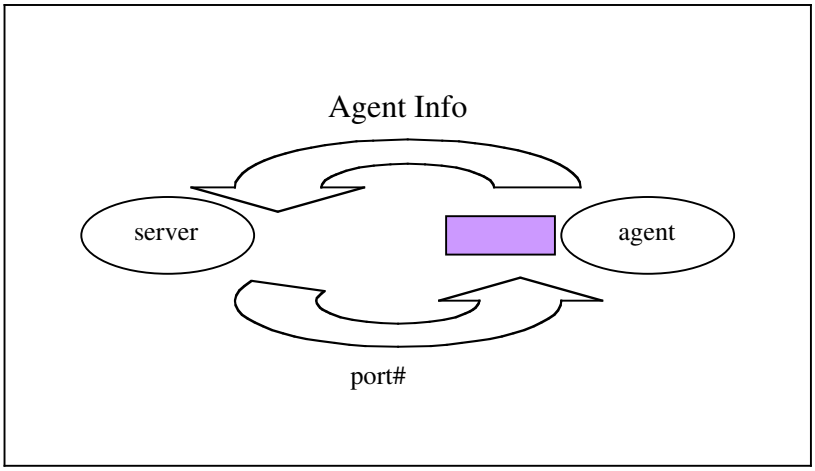


Fig.2 AGENT REGISTRATION



When an agent is created, it immediately registers at the **AgentMonitor** by passing on to it certain defining characteristics about itself. In return, the **AgentMonitor** issues a port number to the agent (Fig.2).

Via this backport the server sends a ping message to check whether the agent is active or not (Fig.4). The **AgentMonitor** is not only capable of monitoring agent viability, but can also detect occurrences of inter-agent communication (Fig.3), and can track the path of a mobile agent as it moves from one host to the other. In order for each group and its members to relocate to a new host to continue operations there, a previously executing **AgentCatcher** must be there to receive them. This causes the agent to be reinitialized at the **AgentMonitor** (Fig.5).

Fig.3 MONITORING AGENT COMMUNICATION

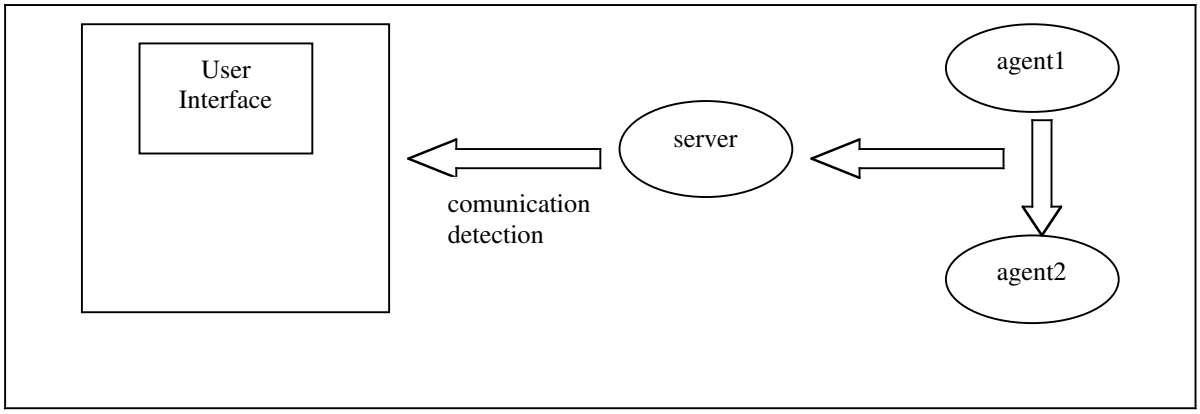


Fig.4 PING ACTIVE AGENTS

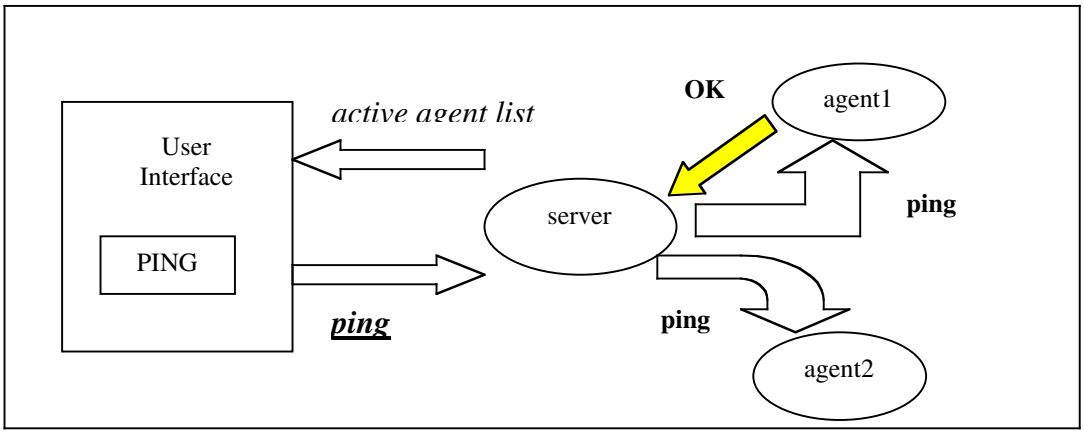
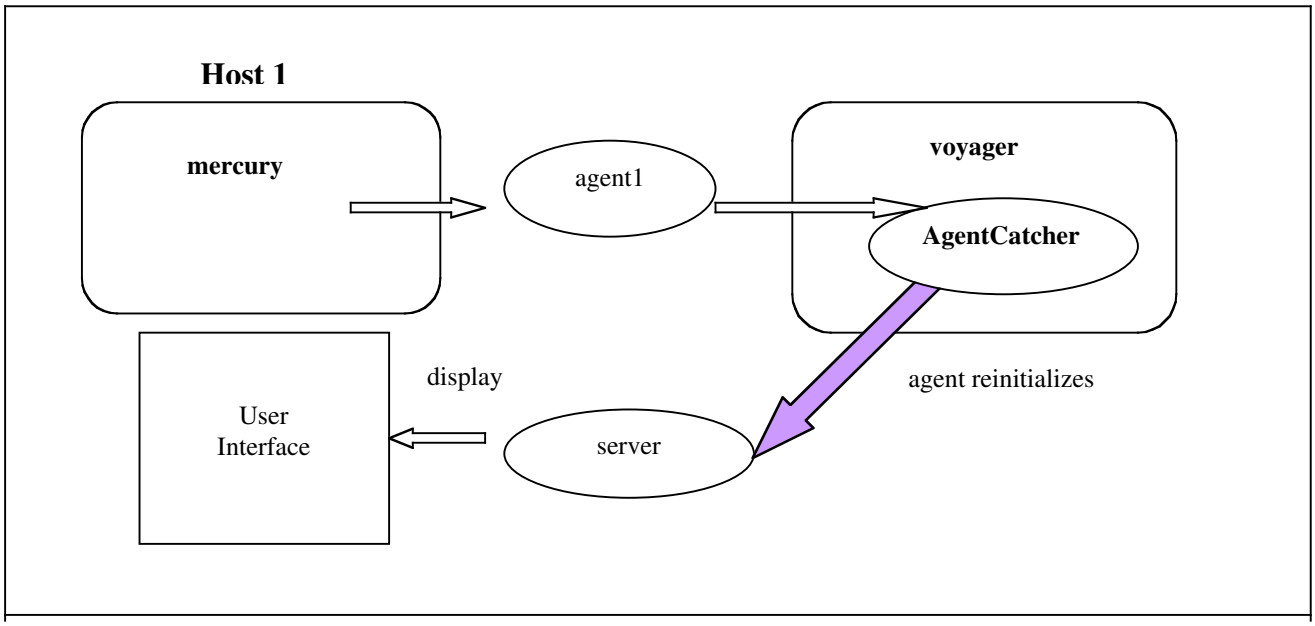


Fig. 5 MONITORING MOBILE AGENTS



Synopsis Of The group Class

Within this synopsis of the group class, members are broken down into four functional groups — **constructors**, **public instance methods**, **private instance methods**, and **private utility groups**. The **constructors** are used to instantiate groups or subclasses of groups. **Instance methods** operate on instances of the class rather than on the class itself. The **public instance methods** provide the user with an interface of seven methods, whilst the **private instance methods** are called within the class itself. The **public** and **private** modifiers show the accessibility of the methods to the user. **Public** methods are accessible to all users, whilst **private** methods are only accessible within the class itself and are never available to the user. Finally, the **private utility methods** are helper functions which are called by other methods within the group class to help them process its data and to efficiently carry out their duties.

public class **group** extends Agent

// Public Constructors
public **group**()
public **group**(String gpname, String membname, String monitor, String ping)

// Public Instance Methods
public void **enter**(Vector gp, String str)
public void **delete**(Vector gp, String str)

public void **merge**(Vector gp, String str)
public void **split**(Vector gp, String str)
public void **destroy**(Vector gp, String str)
public void **move**(Vector gp, String str)
public void **communicate**(Vector gp, String str)

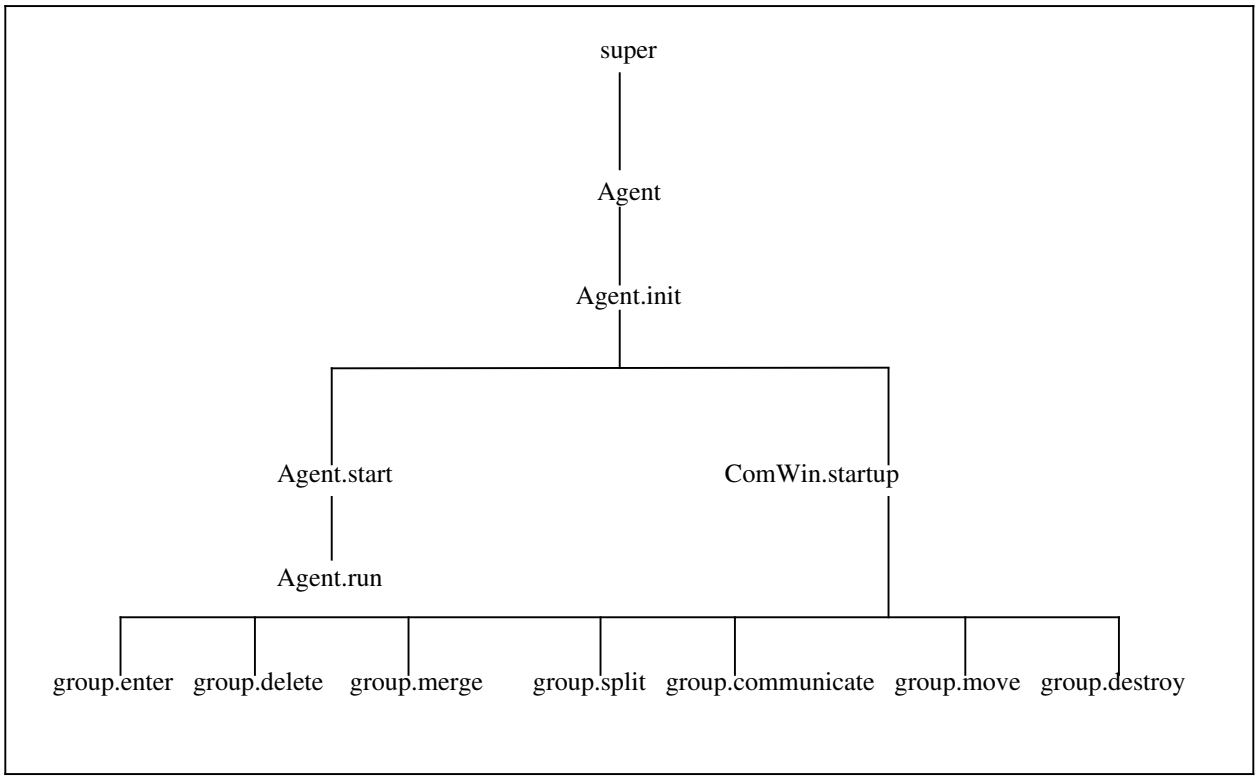
// Private instance methods
private void **register(** String str, int monitorport)
private String **memberConnect**(String membname, int memport)

// Private utility methods
private int **findgpMember**(Object membname)
private Properties **toProperties**(Object proplistst ring)
private Vector **backToVector**(String gpliststring)
private String **getstrToken**(String str)

Details Of Group Class Methods

The chart below shows the function hierarchy within the group class, ending at the level of the public instance methods. Arguments passed to these functions are omitted for purposes of clarity.

Fig.6 FUNCTION HIERARCHY CHART FOR THE GROUP CLASS



A. Constructor

1. group (String gpname, String membname, String monitor, String ping)

To create an instance of a group, the arguments passed to the super constructor match the four String parameters of the **Agent** superclass constructor. They include the groupname, group leader name, monitor name and ping value. Each group, like each agent

- opens a backport via which it will be pinged by the AgentMonitor to ascertain whether the agent is active or inactive,
- calls a *startup* method. Whilst the agent s *startup* method is empty, the group s *startup* method in the **ComWin** class controls a command window.

B. The ComWin class

This public class contains the public instance methods — *startup*(), and *carryon* (). The latter calls *startup*() after the group has relocated and reinitialized itself at the server. Via the command window, the user will be able to send group commands to each group leader. The fields of the user input string are specified in the table below. The commands, formatted in this way, cause the appropriate public group instance method to be invoked.

Fig.7. TABLE SHOWING USER INPUT STRING FORMAT

METHOD	PORT#	COMMAND	ARGUMENT	ARGUMENT	ARGUMENT
enter	agentport#	enter	agentname		
delete		delete	agentname		
merge	agentport#	merge	gpname		
split	agentport#	split	gpname		
destroy		destroy	all		
		destroy	self		
		destroy	agentname		
communicate		communicate	msg	message	
move		move		hostname	port#
		move	self	hostname	port#

B. **Public Instance Methods**

The seven **public instance methods** are all void methods to which are passed two arguments, the current list *gpMembers* and the unprocessed string entered by the user in the command window.

C.1 enter (Vector gp, String str)

- The user string is input into the command window of the group wishing to be entered.
- The local host name is obtained to identify the address of the group leader.
- This input string is parsed to get its fields which are placed in a properties table used to identify each new member. This table has the following five (5) fields

  member=agentname, gpName=gpname, port=agentport#, address=hostname, subgroup=subgpname

  and is unique at each index of the gpMembers list which is implemented with the use of the Vector Class.

- If an entering agent is not a member of a subgroup then its subgroup field is initialized with no .
- If the entering agent is a group with its own members, a member of that group is contacted via its port number and a list of its group members requested.
- Information from this list is added to the new group leader s *gpMembers* list.
- For each of these members two keys of its properties table are updated with values from the new group. The subgroup key, which can have one value or a list as its value, is initialized with the subgroup name. If the member s group is not already
- subgroup, this name becomes the first name in the subgroup list. If it is already a subgroup the subgroup name is concatenated to the end of the list.
- The new *gpMembers* list is then **communicated** to all members of the newly formed group.

Fig. C.1a THE ENTER METHOD

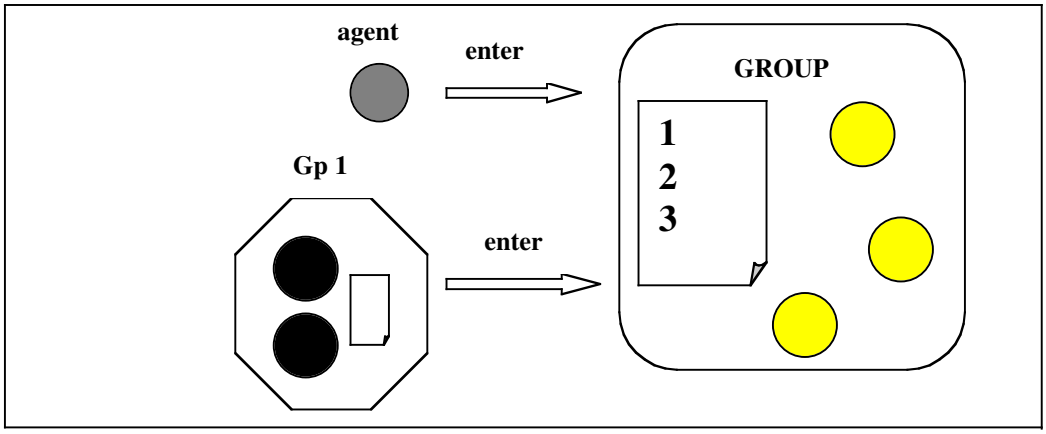
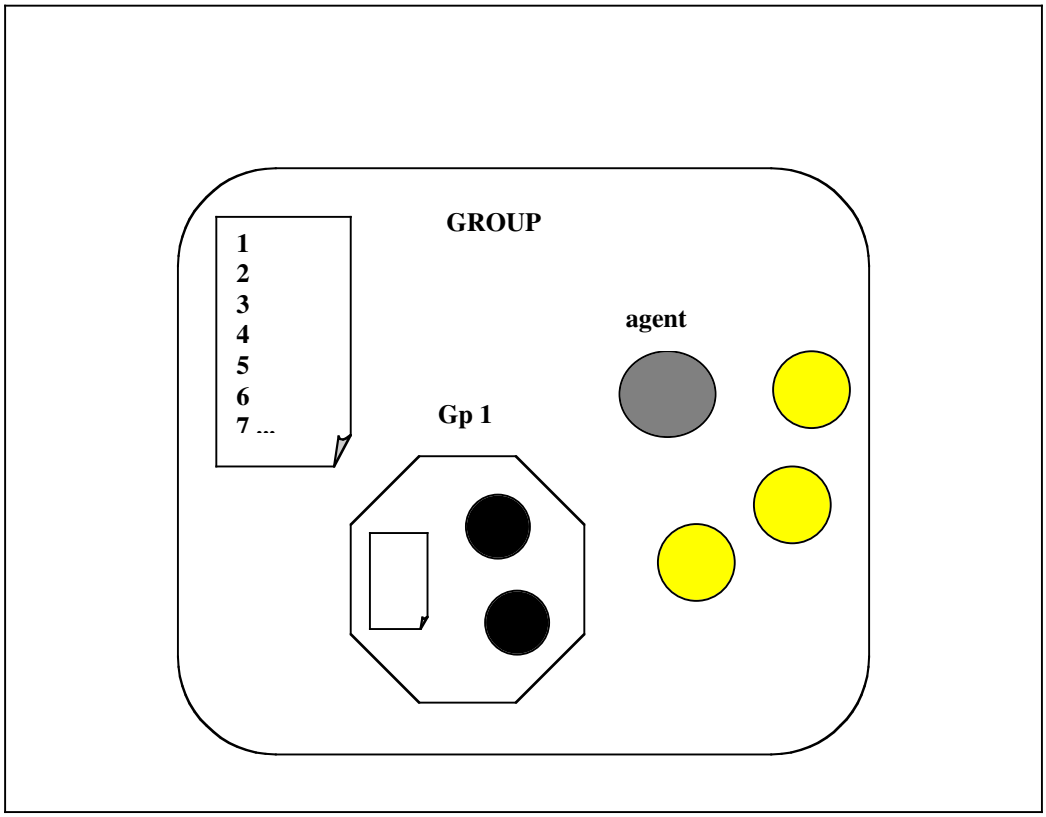


Fig.C.1b THE ENTER METHOD



C.2. delete (Vector gp, String str)

The group leader receives a **delete** command via its command window.

- It checks its *gpMembers* list to see if that agent is present in its group.
- If present, the member s port number is obtained.
- Contact with the member is made via this port# to obtain that member s existing group list.
- The member(s) from this group list that has/have the group leader s name in its/their gpName field is/are deleted and the residual list is returned to it/ them.
- If the member to be deleted from *gpMembers* is a subgroup, all those members which have that subgroup name in their subgroup field are deleted. If the member is an agent, when found it is simply deleted from *gpMembers*.
- The properties table of each of these members that are deleted is added to the special Vector *names*.
- Via the method **communicate** that uses the Vector *names*, the residual lists are broadcast to all agents that are affected.

- The group leader then **communicates** its new *gpMembers* list to any of its remaining members

Fig. C.2a THE DELETE METHOD

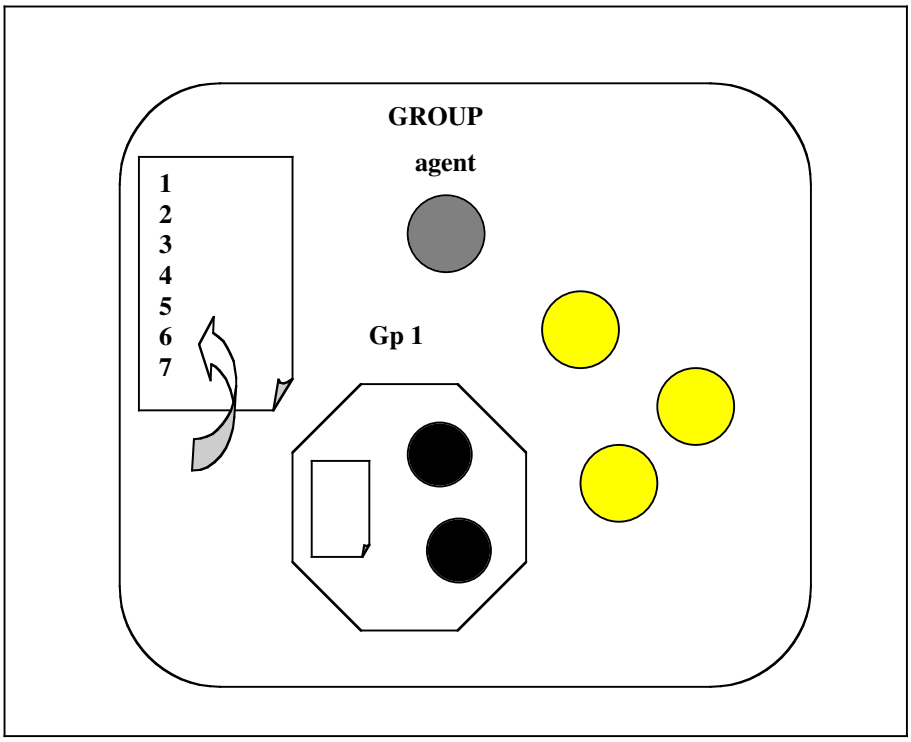
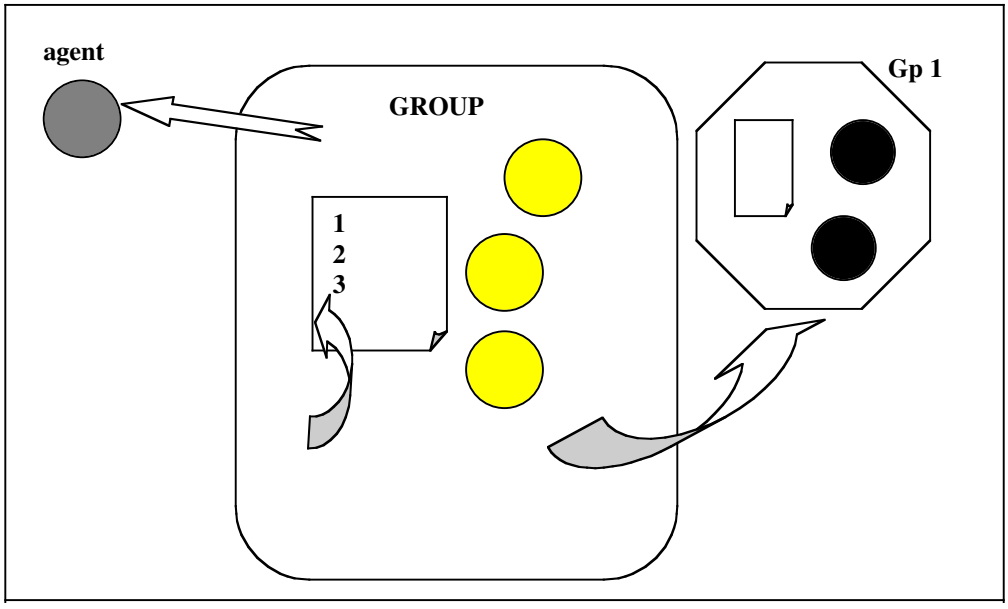


Fig. C.2b THE DELETE METHOD



C.3 merge (Vector gp, String str)

- Again the input string is parsed to obtain the port# for the incoming group, the name of the group to be merged and the name of the group leader .
- Contact via the port is made with a subgroup member and the group list is requested.
- The properties table, used to identify each member of the subgroup, is taken, and its subgroup key is initialized with its group name if the previous value was no . Otherwise, the group name is concatenated to the end of the subgroup key values. The gpName key is updated with the new group leader s name and the properties list is added to the group leader s *gpMembers* list.
- The group leader **registers** this change at the AgentMonitor.
- The new *gpMembers* list is **communicated** to all its members.

Fig. C.3a THE MERGE METHOD

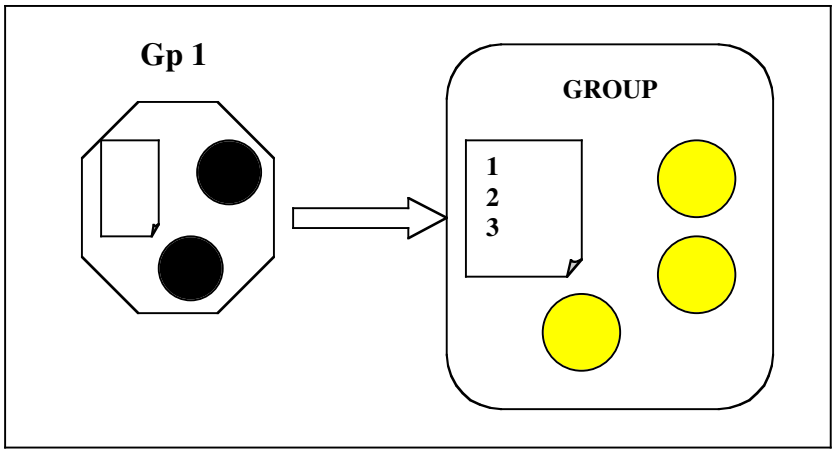
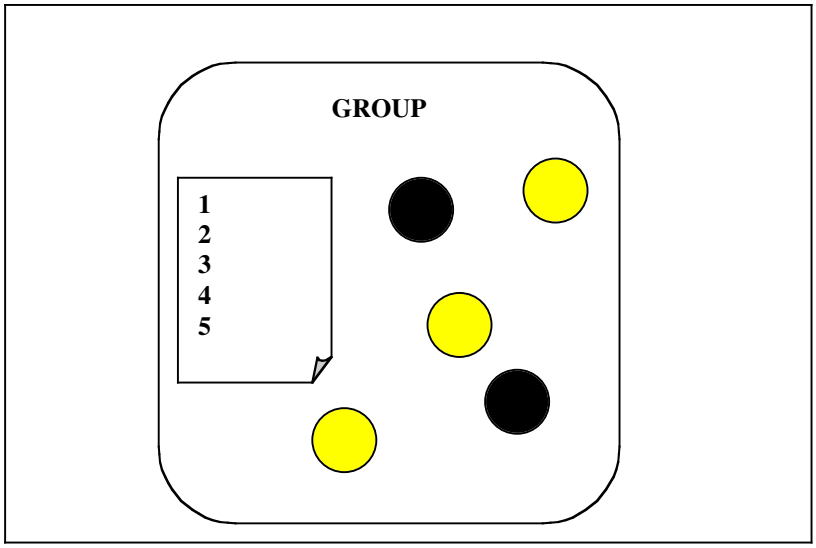


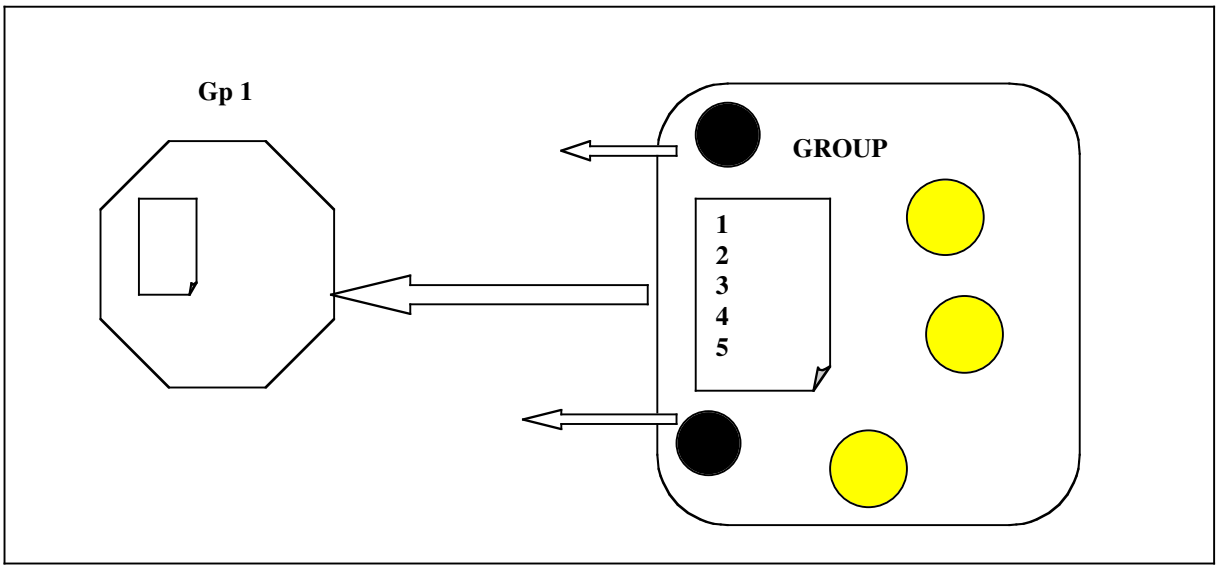
Fig. C.3b THE MERGE METHOD



C.4 split (Vector gp, String str)

- In **split** the input string is parsed to receive each field**.**
- The *gpMembers* list of the group wishing to split away is requested from a member of that group via its port#.
- Any members of that list that had belonged to that subgroup have their subgroup key replaced with no , and their previous gpName returned.
- These members are placed in a new list *names*, the members of which are broadcast to each of them.
- The members are then removed from the group leader s *gpMembers* list.
- The new *gpMembers* list is **communicated** to the remaining members of the group leader s group.

Fig. C.4 THE SPLIT METHOD



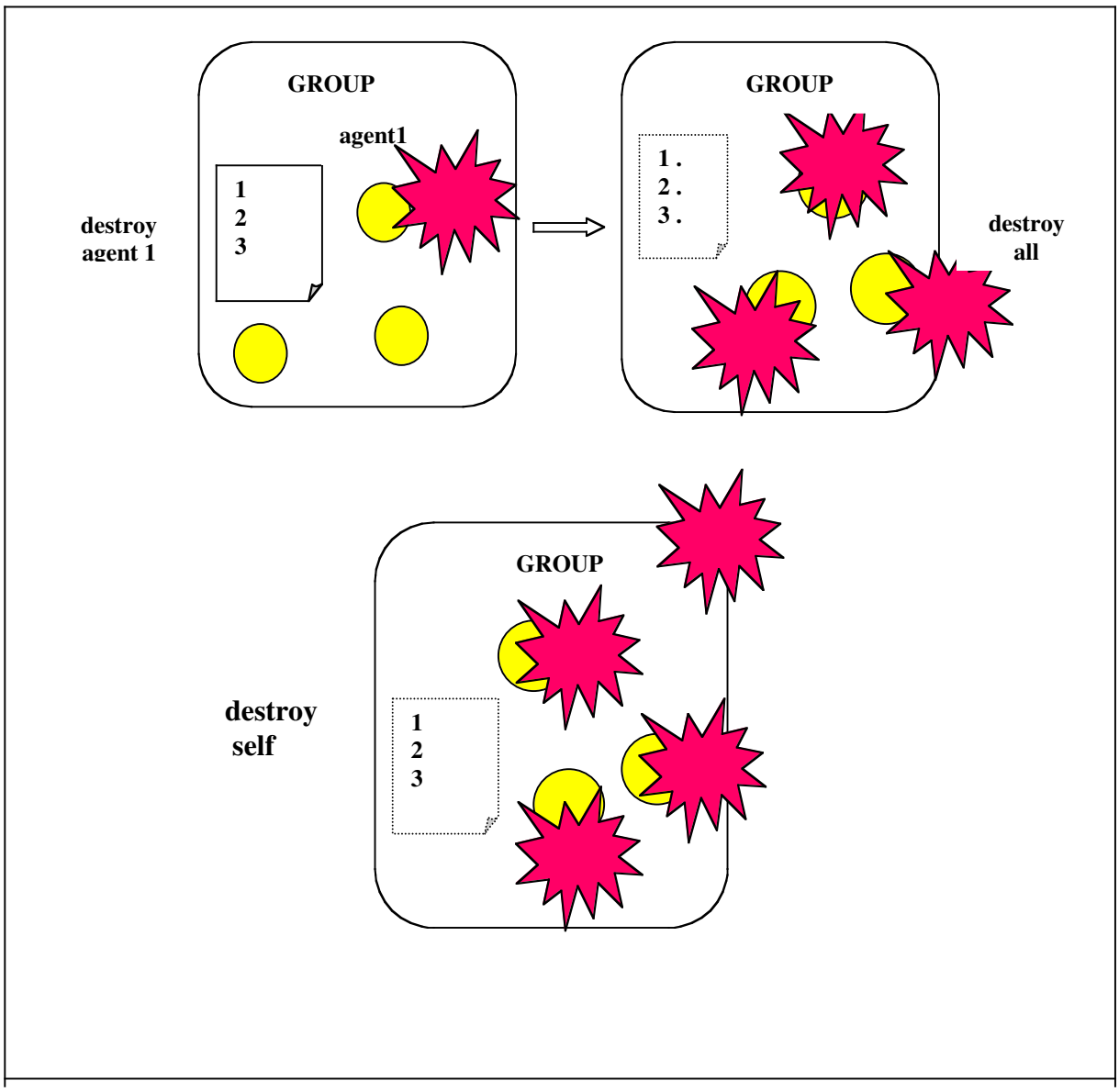
C.5 destroy (Vector gp, String str)

The **destroy** command can have three variant forms as shown in the user string format table given above.

- The string is parsed to obtain its fields.
- If the command is to destroy self the group leader exits from the system.
- If a specific agent has to be destroyed, that agent s port# is obtained from the *gpMembers* list and an exit command is sent to it. This command invokes the system exit procedure in the agent.
- If the agent to be destroyed is a subgroup, all the subgroup members are placed in the special list *names* and the exit message communicated to them.

- The destroyed group member, single agent or subgroup members is/are removed from the *gpMembers* List.
- If the group leader wishes to destroy all its members the exit message is **communicated** to all its members.

Fig. C.5 THE DESTROY METHOD



C.6 move (Vector gp, String str)

- The string is parsed to receive the hostname which is tacked on to the standard address of computers on our local network to give the full remote host address. The port# of the application on the host machine that will receive the moved agents is extracted, and the name of the group name is recorded.
- The group leader updates the address of its *gpMembers* with the new address and **communicates** this new list to them.
- The leader **communicates** the move command to all its members providing them with the remote host address and the port number.
- This move command invokes a **send** procedure possessed by all agents that allows them to move to the waiting host.
- After all its members have relocated to their new host the group leader sends itself across.
- If the agent to be moved is a subgroup leader, a command move self is input into that group leader s command window causing it to send itself across to the indicated host.
- There is no need for the group to register this operation at the AgentMonitor since this is done automatically, after a change of address, when the agent reinitializes itself.

Fig. C.6a THE MOVE METHOD

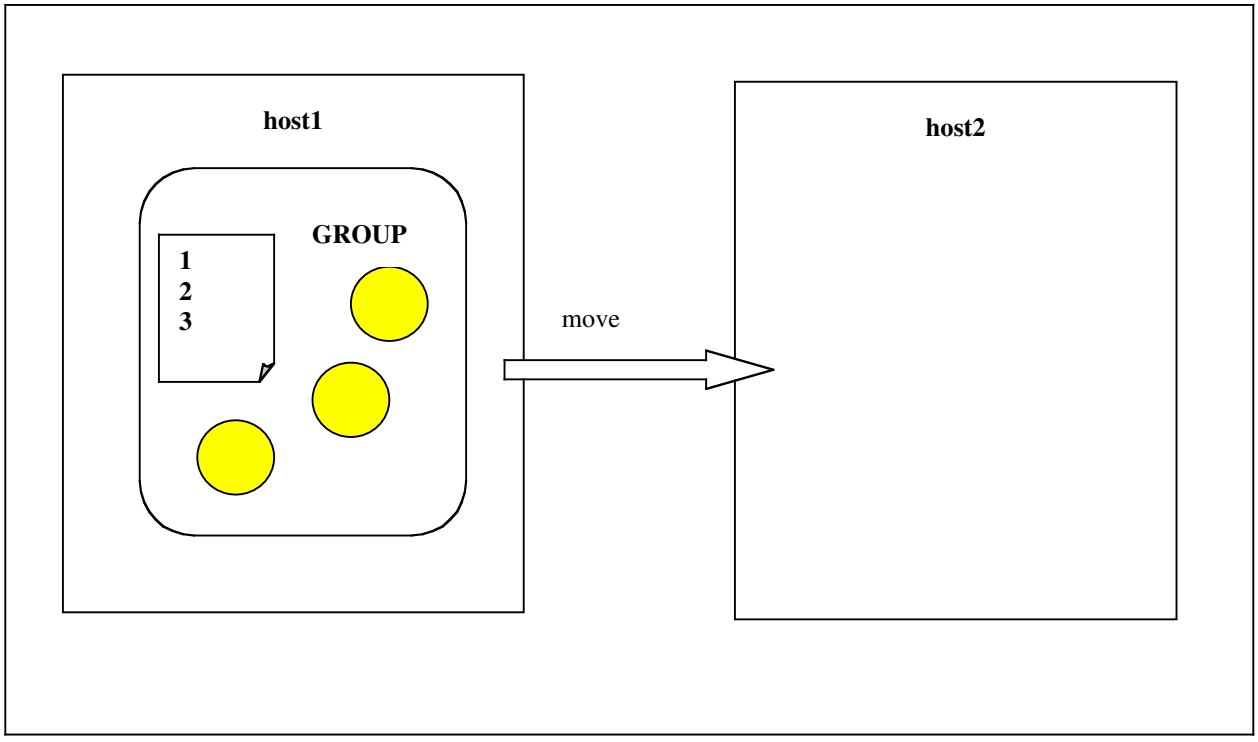


Fig. C.6b THE MOVE METHOD

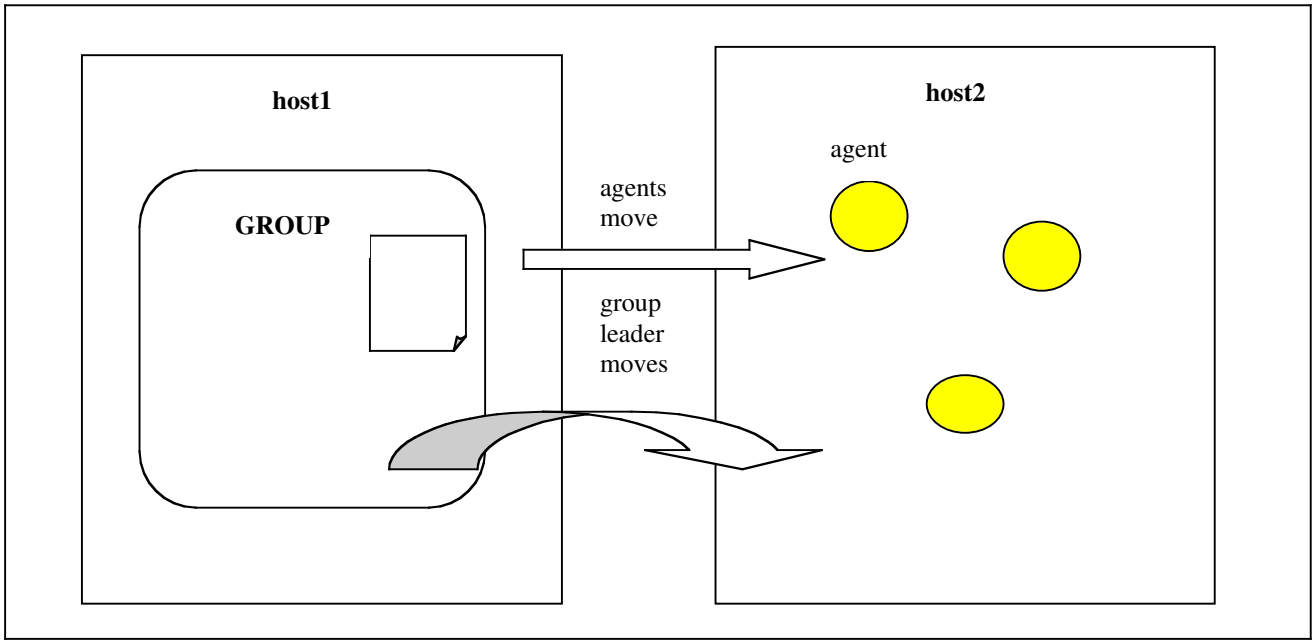
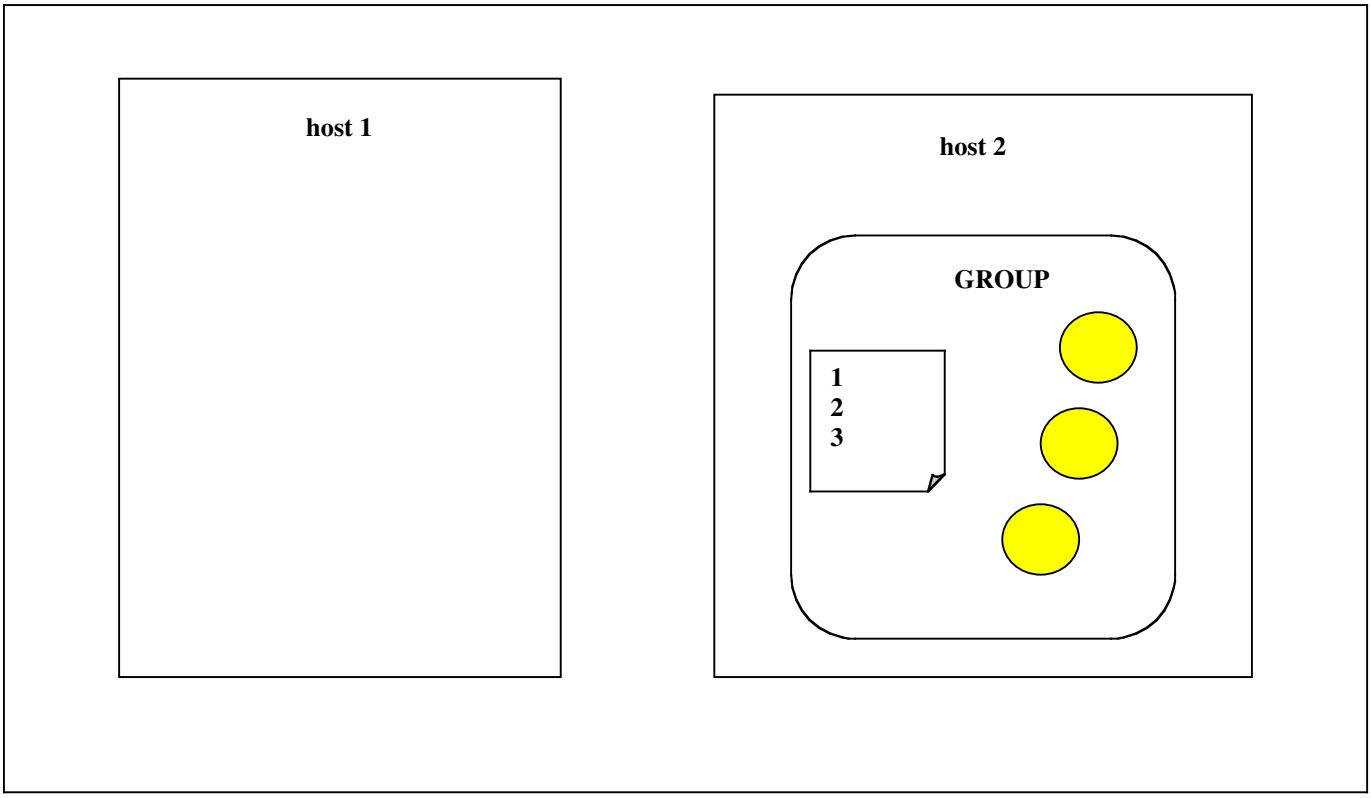


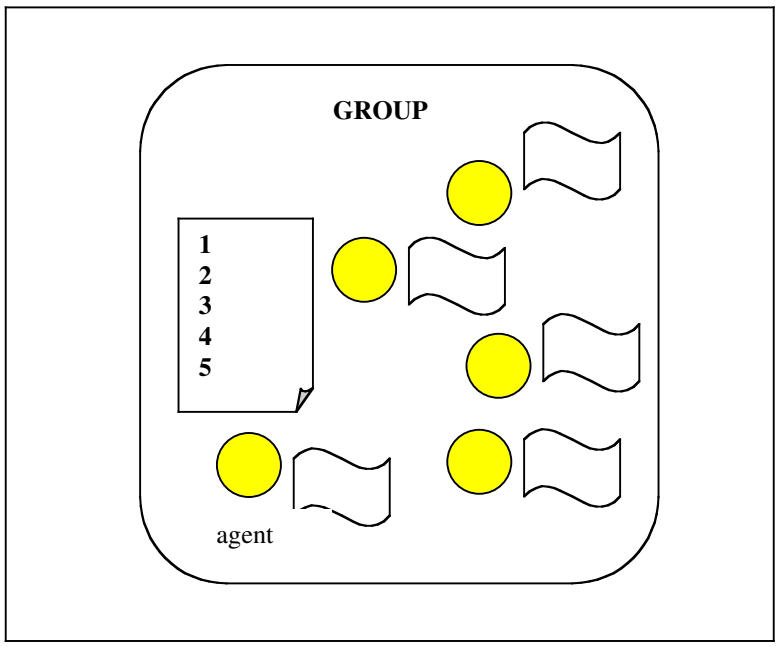
Fig. C.6c THE MOVE METHOD



C.7 communicate (Vector gp, String str)

- The **communicate** method passes its message verbatim to all members populating the *names* list by obtaining their port#s from the list and communicating via these.
- These messages, when received by the agents, may invoke other procedures such as **send**, or System procedures such as **exit** or **out.println.**
  If the message is to be directed to specific members, the names of these members are placed in a vector called *names* which the group leader uses instead of its general *gpMembers* list.

Fig.C.7 THE COMMUNICATE METHOD



D. The Private Instance Methods

The two private instance methods **register** and **memberConnect** require two arguments each — a String and an integer. Whereas **register** is void, i.e returns no value, **memberConnect** returns a String.

D.1. register (String m, int p)

- In **register,** each of the user input strings, minus the first agentport# field, is sent to the **AgentMonitor** via that application s port#. These commands invoke processes in

the AgentMonitor that **enter**, **delet**e, **move**, **merge**, and **split** groups in the *agentList.* A *groupList* is then built from the agentList.

D.2. **memberConnect (String m, int p)**

- When a single member agent has to be contacted and some response is expected the private method **memberConnect** is used.

E. **The Private Utility Methods**

E.1. The **findgpMember (Object membname)** method is passed the name of a member in the list as an Object. If the member is found its index in the list is returned.

E.2. The method which converts a properties table in the form of a String back to a properties table object is **toProperties (Object proplistst**r**ing).**

E.3. A Vector that has been converted to a String is reconverted to its original form using the **backToVector (String gplistst**r**ing)** method which returns a Vector.

E.4. The string input by the user in each group s command window is parsed using the **getstrToken (String str)** that returns the first token of the string passed to it.

Future Work: The present group class provides a skeletal framework upon which many new features will be added. In the future we hope

- To improve the stability of the system and its ability to support a multiplicity of executing threads,
- To extend the functionality of the visual display to reveal, not only the ping and traffic, but also the history of group members,
- To collect subgroup statistics.
- To provide mechanisms by which a group can tell whether there are sufficient resources at a remote location to which it wishes to migrate.
- To build a test bed for the group class.
- To provide mechanisms that would permit group creator proprietary rights to certain public instance group methods such as destroy etc.
- To institute criteria upon which a merger, split, delete, enter, or communicate command is sent to a group member: e.g When should a merger take place? How should it be done and with whom should it be done?
- To be able to associate a detected communication socket with an agent name.
- To extend the use of the group class from a single user to multiple users
- To treat each method as an atomic transaction process and to put mechanisms in place that would allow rollback to take place if a method cannot be completed in its entirety.