Cooperating Mobile Agents

Sukumar Ghosh     Anurag Dasgupta
{ghosh, adasgupt}@cs.uiowa.edu

Abstract

This article discusses four different applications where mobile agents cooperate with one another to accomplish a task in a network of processes. These applications range from topology discovery and distributed data structure implementation to fault diagnosis and system stabilization.

1 Introduction

A mobile agent is a piece of code that migrates from one machine to another. The code (often called the script), which is an executable program, executes at the host machine where it lands. In addition to the code, agents carry data values or procedure arguments or results that need to be transported across machines. Compared to messages that are passive, agents are active, and can be viewed as messengers.

Mobile agents are convenient tools in distributed systems, both at the applications layer and as well as at the middleware level. The promise of mobile agents in bandwidth conservation or disconnected modes of operation is now well accepted. Deploying multiple mobile agents cooperating with one another can add a new dimension to distributed applications. While parallelism is the obvious advantage, the issues of load balancing, agent rendezvous, and fault tolerance play major roles. Among numerous possible applications, we highlight the following four problems, each with a different flavor of cooperation.

Mapping of an unknown network. Network mapping is also known as the topology discovery problem. Making such a discovery using a single mobile agent is equivalent to developing an efficient algorithm for graph traversal. With multiple agents, the challenge is to develop an efficient cooperation mechanism so that the discovery is complete in the fewest number of hops and redundant traversals are avoided.

Concurrent reading and writing. A distributed data structure has different components mapped to host machines at different geographic locations. As multiple agents concurrently access such a distributed data structure, the reading agent the writing agent need to be properly synchronize their operations so that the semantics of data sharing are preserved.

Black hole search. A black hole is a node that can potentially capture a visiting agent and thus, disrupt an application. Although it implies a malicious intent on the part of the host, a black hole can be as simple as a crashed node. If black holes can be located, then traversal paths can be rerouted without incurring further loss of mobile agents.

Stabilization. Transient failures occasionally corrupt the global state of a distributed system, and stabilization is an important technique for restoring normal operation. To stabilize a network, a mobile agent patrols the network, and plays the role of traveling repairperson. Multiple agents can expedite the process of stabilization, but in doing so, some synchronization issues need to be resolved. We will address how multiple agents can be deployed for maximum speedup of a stabilizing application.
This article has seven sections. Section 2 describes the model and the notations. Sections 3 through 6 address the four problems highlighted above. Finally, Section 7 contains some concluding remarks.

# 2 The model

We represent a distributed system by a connected undirected graph $G = (V, E)$, where $V$ is the set of nodes representing processes, and $E$ is the set of edges representing channels for interprocess communication. Basic interprocess communication uses messages that are received in the same order in which they are sent. Processes do not have access to a global clock.

Whenever appropriate, we will represent the program for each process of a set of rules. Each rule is a guarded action of the form $g \rightarrow A$, where $g$ is a boolean function of the state of that process and those of its neighbors received via messages, and $A$ is an action that is executed when $g$ is true. An action by a process involves several steps, these include: receiving a message, updating its own state, and sending a message. The execution of each rule is atomic, and it defines a step of the computation. When more than one rule is applicable, any one of them can be chosen for execution. The scheduler is unfair. A computation is a sequence of atomic steps. It can be finite or infinite.

**Definition.** A global state of the system is a tuple $(s(0), s(1), \ldots, s(n))$, where $s(i)$ is the state of process $i$, together with the states of all channels.

Each agent is launched by an initiator node that is also called the agent’s home. An agent consists of the following six components:

1. The identifier $id$ (also called a label), usually the same as the initiator’s id. The $id$ is unnecessary if there is a single agent, but is essential to distinguish between multiple agents in the same system.

2. The agent program: this program is executed when the mobile agent lands on a host machine.

3. The briefcase $B$ containing a set of data variables. It defines the state of the agent computation, as well as some key results that have to be carried across nodes.

4. The previous process $PRE$ visited by the agent.

5. The next process to visit $NEXT$ that is computed after every hop.

6. A supervisory program for bookkeeping purposes.

Each hop by an agent is completed in zero time. The agent computation is superimposed on the underlying distributed computation executed by the network of processes. The state of the agent is defined by its control and data variables, and the state of the distributed system consists of the local states of all the processes. When an agent executes a step, it changes its own state, and also potentially changes the state of the host on which it executes that step. Unless specified otherwise, the visit of an agent at any node will be treated as an atomic event.

Finally, an agent model that involves multiple agents can be either static or dynamic. In the static model, the number of agents and their homes are known at the beginning of the application, and they remain unchanged throughout the entire life of the application. In the dynamic model, an agent can spawn child agents, or kill them whenever necessary. Unless specified otherwise, we will consider the static model only.
3 Mapping a network

We assume that an agent is trying to construct the complete map of an undirected connected graph. It has to explore all the nodes and edges of the graph starting from some node. When the agent traverses an edge, it explores the corresponding edge and both of its end nodes. During exploration, the agent keeps track of the visited nodes and the edges so that it recognizes them later. In particular, after reaching an already explored node \( v \) incident on an explored edge \( e \), the agent recognizes the location of \( v \) and of the other end of \( e \) on the partial map it constructs. The agent also knows the number of unexplored edges incident on an explored node, but does not know the other ends of these edges.

The goal is to explore all the nodes and the edges of the undirected connected graph with the minimum number of edge traversals. One motivation for visiting all nodes and retrieving data from unknown nodes in a vast network is network maintenance. The agent continuously patrols the network. Also, fault detection in a network requires this type of perpetual exploration. Before we discuss the multi-agent case, we introduce a few exploration protocols using a single mobile agent.

3.1 Exploring Undirected Graphs

The penalty of an exploration algorithm running on a graph \( G = (V, E) \) is the worst-case number of traversals in excess of the lower bound \(|E|\). The total cost of an exploration algorithm running on a graph \( G = (V, E) \) is the worst case number of edge traversal it uses, taken over all starting points and all adversary decisions. The adversary has the power of choosing an arbitrary unexplored edge.

Panaite and Pelc [1] provide an exploration algorithm whose penalty is \( O(|V|) \) for every graph. In fact, they showed that the penalty never exceeds \( 3|V| \). The natural heuristics such as GREEDY and DFS fail to achieve the penalty \( O(|V|) \) for all graphs. The following theorem shows the inefficiencies of these two strategies:

**Theorem 1** The penalties of GREEDY and DFS are not linear in the order of the graph.

At any stage of the algorithm execution, the edges that are already traversed are called explored, and the remaining are called free. A node is saturated if all its incident edges are explored. Otherwise it is free.

The arguments why DFS and GREEDY fail are not hard to see. In both the cases, the agent uses unexplored edges as long as possible, and when stuck at a node \( v \), it uses a simple strategy to reach a free node \( v' \). In case of GREEDY, \( v' \) is the free node closest to \( v \), while in case of DFS, \( v' \) is the most recently visited free node. It turns out that both these choices are too naive. The vision of GREEDY is too local, while DFS does not make sufficient use of the knowledge of the explored subgraph, basing its decisions only on the order of visits.

The algorithm presented by Panaite and Pelc also uses unexplored edges as long as possible. But as opposed to DFS and GREEDY, their algorithm explores the graph in the order given by a dynamically constructed tree. The key difference is the choice of the free node to which the agent relocates after getting stuck. The agent gets back to the node of the dynamically constructed tree at which it interrupted the construction of the tree. In this way the number of traversals through already explored edges is reduced. By following the structure of the dynamic tree, the agent is not distracted from systematic exploration of free nodes situated close to it, which is the case for GREEDY. At the same time, temporal preferences dominate over geographic references that lead to the inefficiency of DFS.

Algorithm EXPLORE in [1] addresses this shortcoming. One of the main features of EXPLORE is agent relocations along a dynamically constructed tree \( T \). Assume that the agent starts at node \( r \) of graph \( G \). At any stage of exploration, let \( H \) denote the known subgraph of \( G \). \( T \) represents a tree.
Program EXPLORE

\[ r := \text{the starting node}; \]
\[ T := (\emptyset, \emptyset); \]
\[ v := r; \]
\[ \text{do } V(T) \neq V(H) \rightarrow \]
\[ \text{SATURATE}(v); \]
\[ \text{EXTEND}(T); \]
\[ v := \text{NEXT}(v); \]
\[ \text{Relocate to } v \text{ along a shortest path in } H \]
\[ \text{od} \]

Figure 1: The algorithm for graph exploration with a single agent.

in \( H \), rooted at \( r \) and connecting only the saturated nodes. An overview of this algorithm is given in Fig 1. We first define the different procedures that are the building blocks of the algorithm.

Procedure SATURATE\((v)\) performs a traversal that starts and ends at \( v \) and saturates \( v \). Procedure EXTEND\((T)\) constructs the new tree \( T \) that corresponds to the current \( H \). Procedure NEXT\((v)\) is defined as follows: If \( V(T) = V(H) \) then NEXT\((v)\) returns the node \( v \). Otherwise, it returns a node \( u \in Ext(w) \), where \( w \) is the first node \( w' \) in dfs\((T)\) with \( Chi(w') = \emptyset \) and \( Ext(w') \neq \emptyset \). Here, \( Chi(w') \) denotes the set of children of \( w' \), i.e., neighbors of \( w' \) in \( T \) different from the parent of \( w' \) if it exists, and \( Ext(w') \) denotes the set of extensions of \( w' \), i.e., neighbors of \( w' \) in \( H \) but not in \( T \).

Panaite and Pelc proved the following theorem:

**Theorem 2** The penalty of the proposed algorithm EXPLORE is linear in the order of the graph. It uses \( m + O(n) \) edge traversals, for every graph with \( n \) nodes and \( m \) edges.

### 3.2 Optimal Graph Exploration

For a given graph \( G \) and a given starting node \( v \), a measure of the quality of an exploration algorithm \( A \) is the ratio \( C(A, G, v) / opt(G, v) \) of its cost to that of the optimal algorithm having complete knowledge of the graph. Here \( opt(G, v) \) is the length of the shortest covering walk which is the exploration with fewest edge traversals starting from node \( v \). The cost \( C(A, G, v) \) is the worst case number of edge traversals taken over all of the choices of the adversary. The ratio represents the relative penalty paid by the algorithm for the lack of knowledge of the environment. For a given class \( U \) of graphs, the number

\[ O_U(A) = \sup_{G \in U} \max_v \in G \left( \frac{C(A, G, v)}{opt(G, v)} \right) \]

is called the overhead of algorithm \( A \) for the class \( U \) of graphs. For a fixed scenario, an algorithm is called optimal for a given class of graphs, if its overhead for this class is minimal among all exploration algorithms working under this scenario.

Dessmark and Pelc [2] presented optimal exploration algorithms for several classes of graphs. They considered the following three scenarios:

1. The agent has no a priori knowledge of the graph. They called it exploration without a map.
2. The agent has an unlabeled isomorphic copy of the graph. This is called an unanchored map of the graph.


<table>
<thead>
<tr>
<th>Scenario</th>
<th>Anchored map</th>
<th>Unanchored map</th>
<th>No map</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>Overhead = $\frac{1}{3}$, optimal</td>
<td>Overhead = $\sqrt{3}$, optimal</td>
<td>DFS</td>
</tr>
<tr>
<td>Trees</td>
<td>Overhead = $\frac{1}{3}$, optimal</td>
<td>Overhead &lt; 2</td>
<td>Overhead = 2, lower bound = $\sqrt{3}$</td>
</tr>
<tr>
<td>General graphs</td>
<td>DFS, Overhead = 2, optimal</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Summary of results about optimal exploration (taken from [2])

Algorithm Anchored Line

Let $x = 3a + n$ and $y = 2n - a$

if $x \leq y$ then
    go at distance $a$ in one direction, or until an endpoint is reached;
    if an endpoint is reached
        then return, go to the other endpoint, and stop
    else return, go to the endpoint, return, go to the other endpoint, and stop
else go to an endpoint, return, go to the other endpoint, and stop

Figure 3: Optimal algorithm for exploration on a line graph with an anchored map.

3. The agent has an unlabeled isomorphic copy of the explored graph with a marked starting node. This is called an anchored map of the graph. This scenario does not give the agent any sense of direction, since the map is unlabeled. For example, when the agent starts the exploration of a line, such a map gives information about the length of the line and distances from the starting node to both ends, but does not tell which way is the closest end.

In all scenarios, the assumption is that nodes have distinct labels, and all edges at a node $v$ are numbered $1, \cdots, \deg(v)$ in the explored graph. Otherwise it is impossible to explore even the star graph with three leaves, as after visiting the second leaf, the agent cannot distinguish the port leading to the first visited leaf from that leading to the unvisited one. Hence the agent can recognize the already visited nodes and traversed edges. However, it cannot tell the difference between yet unexplored edges incident on its current position. The actual choice of such unexplored edges is made by the adversary when worst-case performance is being considered. Fig 2 summarizes the main results of [2]

The table indicates that for the class of all undirected connected graphs, DFS is an optimal algorithm for all scenarios except for trees. Without any knowledge, DFS is optimal for trees. Under the scenario with an anchored map, the optimal overhead is at least $\sqrt{3}$ but strictly below 2. Thus DFS is not optimal in that case. [2] gives an optimal algorithm for trees with an anchored map and shows that its overhead is $\frac{2}{3}$. Of the many algorithms described in [2] we choose the simplest case of exploration on lines with an anchored map. This means the agent knows the length $n$ of the line, as well as the distances $a$ and $b$ between the starting node and the endpoints. The algorithm for $a \leq b$ is given in Fig 3.

Fraigniaud et al. [3] showed that for any $K$-state agent and any $d \geq 3$, there exists a planar graph of maximum degree $d$ with at most $K + 1$ nodes that the agent cannot explore. They also showed that in order to explore all graphs of diameter $D$ and maximum degree $d$, an agent needs $\Omega(D \log d)$ memory bits, even if the exploration is restricted to planar graphs. This latter bound is tight. So the worst case space complexity of graph exploration is $\Theta(D \log d)$ bits.
3.3 Collective Tree Exploration with multiple agents

So far we have considered graph exploration with the help of a single agent. In [5], Fraigniaud et al. addressed the problem of collective graph exploration using multiple agents: they considered exploring an $n$-node tree by $k$ agents ($k > 1$), starting from the root of the tree. The agents return to the starting point at the end of the exploration. Every agent traverses any edge in unit time, and the time of collective exploration is the maximum time used by any agent. Even when the tree is known in advance, scheduling optimal collective exploration turns out to be NP-hard. The main communication scenario adopted in the paper is the following: agents write at the currently visited node the information they previously acquired, and they read information (provided by other agents) available at this node. The paper provides an exploration algorithm for any tree with overhead $O\left(\frac{1}{\log k}\right)$. The authors prove that if agents cannot communicate at all, then every distributed exploration algorithm works with overhead $\Omega(k)$ for some trees.

The model for $k$-agents scenario is a little different than that of what described earlier for a single agent. The agents have distinct identifiers but apart from that, they are identical. Each agent knows its own identifier and follows the same exploration algorithm which has the identifier as a parameter. The network is anonymous as before, i.e., nodes are not labeled, and ports at each node have only local labels that are distinct integers between 1 and the degree of the node. At every exploration step, every agent either traverses an edge incident on its current position, or remains in the current position. An agent traversing an edge knows local port numbers at both ends of the edge.

The communication scenario is termed as exploration with write-read communication. In every step of the algorithm every agent performs the following three actions: (a) It moves to an adjacent node. (b) It writes some information in it. (c) It then reads all information available at this node, including its degree. Alternatively, an agent can remain in the current node, in which case it skips the writing action.

Actions are assumed to be synchronous: if $A$ is the set of agents that enter $v$ in a given step, then first all agents from $A$ enter $v$, then all agents from $A$ write, and then all agents currently located at $v$ (those from $A$ and those that have not moved from $v$ in the current step) read. Two extreme communication scenarios are discussed. In case of exploration without communication, the agents are oblivious of one another. At each step, every agent knows only the route it traversed until that point, and the degrees of all nodes it visited. In case of exploration with complete communication, all agents can instantly communicate at each step. In both scenarios, an agent does not know the other endpoints of unexplored incident edges. If an agent decides to traverse such a new edge, the choice of the actual edge belongs to the adversary, when the worst-case performance is being considered.

The exploration algorithm described in the paper has an overhead of $O\left(\frac{1}{\log k}\right)$. To be precise, the algorithm explores any $n$-node tree of diameter $D$ in time $O(D + \frac{n}{\log k})$. The algorithm is described for the stronger scenario first, i.e. for exploration with complete communication. It can be simulated in the write-read model, without changing time complexity. The paper shows that any algorithm must have overhead at least $(2 - 1/k)$ under the complete communication scenario. In order to get overhead sublinear in the number of agents, some communication is necessary. Exploration without communication does not allow any effective splitting of the task among agents.

We outline here the algorithm for exploration with complete communication as described in [5]. Let $T_u$ be the subtree of the explored tree $T$, rooted at node $u$. $T_u$ is explored, if every edge of $T_u$ has been traversed by some agent. Otherwise, it is called unexplored. $T_u$ is finished, if it is explored, and either there are no agents in it, or all agents in it are in $u$. Otherwise, it is called unfinished. $T_u$ is inhabited, if there is at least one agent in it. Fig 4 shows the algorithm.
Fix a step \( i \) of the algorithm and a node \( v \) in which some agents are currently located.

There are three possible (exclusive) cases.

{Case 1} Subtree \( T_v \) is finished.

(Action) if \( v \neq r \), then all agents move from \( v \) to the parent of \( v \), else all agents from \( v \) stop.

{Case 2} There exists a child \( u \) of \( v \) such that \( T_u \) is unfinished.

Let \( u_1, u_2, \ldots, u_j \) be children of \( v \) for which the corresponding trees are unfinished, ordered in increasing order of the local port numbers of \( v \). Let \( x_l \) be the number of agents currently located in \( T_{u_l} \). Partition all agents from \( v \) into sets \( A_1, \ldots, A_j \) of sizes \( y_1, \ldots, y_j \), respectively, so that integers \( x_l + y_l \) differ by at most 1. The partition is done in such a way that the indices \( l \) for which integers \( x_l + y_l \) are larger by one than for some others, form an initial segment \((1, \ldots, z)\) in \((1, \ldots, j)\). Moreover, sets \( A_l \) are formed one-by-one, by inserting agents from \( v \) in order of increasing identifiers.

(Action) All agents from the set \( A_l \) go to \( u_l \), for \( l = 1, \ldots, j \).

{Case 3} For all children \( u \) of \( v \), trees \( T_u \) are finished, but at least one \( T_u \) is inhabited.

(Action) All agents from \( v \) remain in \( v \).

Figure 4: Algorithm for collective exploration with complete communication.

### 3.4 Deterministic Rendezvous in Arbitrary Graphs

The rendezvous problem is defined as follows: two mobile agents located in nodes of an undirected connected network have to meet at some node of the graph. If nodes of the graph are labeled, then agents can decide to meet at a predetermined node, and the rendezvous problem reduces to graph exploration. However, in many practical applications where rendezvous is needed in an unknown environment, such unique labeling of nodes may not be available, or limited sensory capabilities of the agents may prevent them from perceiving such labels. As before we assume that the ports at a node are locally labeled as \((1, 2, \ldots, d)\) where \( d \) is the degree of the node.

Agents move in synchronous rounds. In each round, an agent may either remain in the same node, or move to an adjacent node. Agents can start up simultaneously or arbitrarily, i.e., an adversary can decide the starting times of the agents.

One assumption of deterministic rendezvous is, if agents get to the same node in the same round, they become aware of it, and rendezvous is achieved. However, if they cross each other along an edge, moving in the same round along the same edge in opposite directions, they do not notice each other. So rendezvous is not possible in the middle of an edge. The time used by a rendezvous algorithm, for a given initial location of agents in a graph, is the worst-case number of rounds since the startup of the later agent until rendezvous is achieved. The worst case is taken over all adversary decisions, and over all possible startup times (decided by the adversary) in case of the arbitrary startup scenario.

Each agent knows its own label, but does not know the label of the other agents. If agents are identical and execute the same algorithm, then deterministic rendezvous is impossible even in the simplest case when the graph consists of two nodes joined by an edge. If both agents knew each other’s labels, then the problem can be reduced to that of graph exploration. The same thing applies if the graph has a distinguished node.
The rendezvous problem in graphs has mostly been studied using randomized methods. Dessmark, Fraigniaud and Pelc [6] addressed deterministic algorithms for the rendezvous problem, assuming that agents have distinct identifiers and are located at nodes of an unknown anonymous connected graph. Their paper showed that rendezvous can be completed in optimal time $O(n + \log l)$ on any $n$-node tree, where $l$ is the smaller of the two labels. The result holds even with arbitrary startup. But trees are a special case from the point of view of the rendezvous problem, as any tree has either a central node or a central edge\(^1\), which facilitates the meeting. This technique used for trees cannot be applied to graphs containing cycles.

With simultaneous startup, optimal time of rendezvous on any ring is $\Theta(D \log l)$ and [6] describes an algorithm achieving that time, where $D$ is the initial distance between agents. With arbitrary startup, $\Omega(n + D \cdot \log l)$ is a lower bound on the time required for rendezvous on an $n$-node ring. The paper presents two rendezvous algorithms for the ring with arbitrary startup: an algorithm working in time $O(n \log l)$, for known $n$, and an algorithm polynomial in $n$, $l$ and the difference between the startup times, when $n$ is unknown. The paper also gives an exponential cost algorithm for general graphs, which is later improved. The next section discusses the issue.

### 3.5 Polynomial Deterministic Rendezvous in Arbitrary Graphs

Deterministic rendezvous has previously been shown to be feasible in arbitrary graphs [6] but the proposed algorithm had cost exponential in the number $n$ of nodes and in the smaller identifier $l$, and polynomial in the difference $\tau$ between startup times. The main result of the paper by Kowalski and Pelc [7] is a deterministic rendezvous algorithm with cost polynomial in $n$, $\tau$ and $\log l$. Kowalski and Pelc’s algorithm contains a non-constructive ingredient: agents use combinatorial objects whose existence is proved using a probabilistic method. Nevertheless their rendezvous algorithm is deterministic. Both agents can find separately the same combinatorial object with desired properties, which is then used to solve the rendezvous algorithm. This can be done using a brute force exhaustive search that may be quite complex, but their model only counts the moves of the agents and not the computation time of the agents. The paper concludes with the open question:

Does there exist a deterministic rendezvous algorithm whose cost is polynomial in $n$ and $l$ (or even in $n$ and $\log l$) but independent of $\tau$?

### 3.6 Asynchronous Deterministic Rendezvous in Graphs

Marco et al. [8] studied the asynchronous version of rendezvous problem. Note that in the asynchronous setting, meeting at a node (which is normally required in rendezvous) is in general impossible. This is because even in a two-node graph, the adversary can desynchronize the agents, and make them visit nodes at different times. This is why, the agents are allowed to meet inside an edge as well.

For the case where the agents are initially located at a distance $D$ on an infinite line, the paper describes a rendezvous algorithm with cost $O(D |L_{min}|^2)$ where $D$ is known and $O(D + |L_{max}|)^3$ if $D$ is unknown, where $|L_{min}|$ and $|L_{max}|$ are the lengths of the shorter and longer labels of the agents, respectively. The authors also describe an optimal algorithm of cost $O(n |L_{min}|)$, if the size $n$ of the ring is known, and of cost $O(n |L_{max}|)$, if $n$ is unknown. For arbitrary graphs, they show that rendezvous is feasible if an upper bound on the size of the graph is known. They present an optimal algorithm of cost $O(D |L_{min}|)$ when the topology of the graph and the initial positions of the agents are known to each other. The paper asks two open questions:

1. Is rendezvous with cost $O(D |L_{min}|)$ possible for a ring of unknown size?
2. Suppose that a bound $M$ on the number of nodes of the graph is known to both

---

\(^1\)Every tree has one or two centers. In the latter case, the edges joining the two centers serve as a central edge.
agents. Is there a rendezvous algorithm polynomial in the bound $M$ and in the lengths of the agents’ labels?

4 Concurrent reading and writing

This problem addresses the implementation of a read-write object on a wide area network. The various components of the object are mapped to different processes over the network. We assume that the reading and writing of the global states of a distributed system are carried out by reading agents and writing agents respectively. The notion of a consistent snapshot is available from Chandy and Lamport’s seminal paper [10]. Also, Arora and Gouda [12] illustrated how to reset a distributed system to a predefined global state. None of these used mobile agents. Our goal is to not only examine how a single reading or writing agent can perform these operations, but also to illustrate how reading and writing operations can be performed when multiple agents are active in the network at the same time (Fig 5). Clearly, we plan to treat the network of processes as a concurrent object that can be accessed by the read and the write operations. There are numerous possible applications. The reading and the writing agents can cooperate with one another to implement a consistency model on a distributed data structure. Multiple reading agents may work simultaneously to speed up data retrieval. Multiple reading and writing agents may work together to expedite fault recovery. In parallel programming languages like Linda [24], processes communicate with one another via a tuple space. On a network of processes, the tuple space is a part of the network state that can be concurrently read or written by two or more agents. In the area of electronic commerce, multi-agent protocols are rapidly growing.

![Figure 5: A reading and a writing agent traversing a network.](image)

The classical approach to implementing concurrent objects consists of the use of critical sections and a mechanism to mutual exclusion. This approach involves waiting, since at most one process is allowed in the critical section at any time. A side-effect is deadlock – if one process is stuck in the critical section, then it indefinitely blocks all processes waiting to enter the critical section. For asynchronous, fault-tolerant distributed systems, it is much more desirable to design a non-blocking or wait-free [18] solution, where each access by an agent is completed in a finite number of steps, regardless of the speeds of other agents or processes. In this article, we will explore non-blocking solutions to the problem of concurrent reading and writing problem, using the agent model. The basic correctness condition relevant to our solutions is that even though the operations of reading and writing by two different agents may overlap, the results of the reading must insure as if the operations are atomic, and consistent with what was written by the writing agent(s). Readers might
note that similar algorithms have already been proposed for shared registers by Lamport [20], where he showed how to implement atomic registers using weaker versions of registers where the reading and writing operations overlap in time. Below we clarify the semantics of overlapped read and write.

Assume that initially $\forall i, s(i) = 0$. Let $W^k$ and $W^{k+1}$ denote two consecutive write operations by the writer: $W^k$ updates the local state of every process to 6, and $W^{k+1}$ updates the local state of every process to 5. Let the reading agent take three successive snapshots $R^i$, $R^{i+1}$ and $R^{i+2}$ in overlapped time as shown in Fig 6.

1. Each read operation must return a consistent value of the global state, that will correspond to (i) the previous reset state before the read started, or (ii) the state to which the system is being currently set, (iii) or a state reachable from one of these.

2. No read returns a global state that is “older than”, i.e. causally ordered before, the global state returned by the previous read.

As a consequence of the second condition, it is okay if both $R^i$ and $R^{i+1}$ return a state $(0, 0, 0, ...)$ or a state reachable from it, but it is not acceptable for $R^{i+2}$ to return $(6, 6, 6, ...)$ and $R^{i+1}$ to return $(5, 5, 5, ...)$. Note that similar conditions are true for atomic registers [21] too.

\[
\begin{array}{c}
W.k \\
writes 6 in each process
\end{array}
\ 
\begin{array}{c}
W.k+1 \\
writes 5 in each process
\end{array}

\begin{array}{c}
R.i \\
R.i+1 \\
R.i+2
\end{array}

Figure 6: Atomic behavior when multiple reads overlap a write.

4.1 The 1-reader and 1-writer cases

Before presenting the protocol of concurrent reading and writing, we first describe the individual protocols in the single-reader and the single-writer cases. In each case, the agent performs a DFS traversal of the network infinitely often using the protocol described in [15]. The briefcase of the agent is initialized by its home, and at each visited node, the agent executes its designated program before hopping to the next node designated by $NEXT := DFS$. Ideally, the processes should be oblivious to when the agent traverses and performs the read or write operations. However, in the single-reader case (and in general with the static model of agents), a little help from the visited processes seems unavoidable.

4.2 1-reader protocol

The snapshot is taken by the reading agent and saved in its briefcase. The agent starts by taking a snapshot of its home process. Thereafter, as the agent visits a node, it records a snapshot state, which ordinarily is the local state of that node. When the agent returns home after each complete traversal, it computes a consistent global state of the system from the states recorded at the individual nodes.

For simplicity, we assume that the channels have zero capacity, so the channel states are irrelevant. At any moment, a message circulating in the system can be of one of the following four types: (1) from an unvisited node to an unvisited node (2) from a visited node to a visited node (3) from a visited node to an unvisited node (4) from an unvisited node to a visited node. Of these, when
a message $m$ propagates from a visited node $i$ to an unvisited node $j$, there is the potential for a causal ordering between the recordings of $s(i)$ and $s(j)$ by the agent. This is because the following causal chain (record $s(i) <$ send $m <$ receive $m <$ record $s(j)$) may exist. To avoid this, we have to ask process $j$ receiving $m$ to save its current state $h(j)$, before receiving $m$. It is this saved value that will be returned to the agent as the local state of process $j$. To detect a message from a visited to an unvisited node, we use the following mechanism:

Each reading agent tags its traversal with a sequence number $SEQ \in \{0,1,2\}$ that is stored in its briefcase. Before each traversal begins, the home increments this value modulo 3. Each node has two variables $seq$ and $agent\_seq$ both of which are updated to_SEQ when the agent visits that node. The value of $agent\_seq$ is appended to every message sent out by a node. Accordingly, when a visited node $i$ sends a message $m$ to an unvisited node $j$, the condition $agent\_seq(i) = agent\_seq(j) \oplus 3 1$ will hold. When node $j$ receives the message, it sets $seq(j)$ to $-1$ and also sets $agent\_seq(j)$ to the value of $agent\_seq(i)$, and saves $s(j)$ into the a history variable $h(j)$. Subsequently, when the reading agent visits $j$, $seq(j)$ will be reset to the value of $SEQ$ in the briefcase of the agent, and the history variable is deallocated. This leads to the program in Fig 7:

Program for the agent while visiting process $i$

agent variables $SEQ$, $S$; ($S.i$ records the state $s(i)$ of node $i$)

process variables $seq$, $s$, $agent\_seq$, $h$ (initially $h$ is empty);

if $SEQ = agent\_seq(i) \oplus 3 1 \land seq(i) \neq -1$ →
   $seq(i) := SEQ$; $agent\_seq(i) := SEQ$; $S.i := s(i)$;
   $\square$ $SEQ = agent\_seq(i) \land seq(i) \neq -1$ → skip
   $\square$ seq$ = -1$ → $seq(i) := SEQ$; $S.i := h(i)$; delete $h(i)$
fi:
$NEXT := DFS$

Program for process $i$

do true →
   if message from $j$: $agent\_seq(j) = agent\_seq(i) \oplus 3 1$ →
      $h(i) := s(i)$; $seq(i) = -1$; $agent\_seq(i) := agent\_seq(j)$;
   fi:
      accept the message;
   od

Figure 7: The 1-reader protocol

We present without proof the following lemma. The proof can be found in [17].

Theorem 3 Each reading of the global state returns a consistent global state, and if two readings are consecutively taken by the reading agent, then the second read cannot return a reading older than the value returned by the first read.

4.3 1-writer protocol

For the writing protocol, we preserve the consistency of the reset operation by disallowing all messages between nodes whose states have been updated and the nodes whose states are yet to be updated [12]. To distinguish between consecutive write operations, we introduce a non-negative

2This flags the agent to record the state from the history.
Program for the writing agent while visiting process $i$

The agent wants to reset the global state to $W$

agent variables CLOK, $W$;

process variables clock, $s$;

if $(\text{clock}(i) < \text{CLOCK})$ → 
  $s(i) := W_i$;
  $\text{clock}(i) := \text{CLOCK}$

fi

$\text{NEXT} := \text{DFS}$

Figure 8: The 1-writer protocol

integer variable CLOK with the writing agent. For the purpose of reset only, a binary value of CLOK will suffice. However, CLOK will need to have more than two values when we address concurrent reading and writing in the next section. The write will update the value of clock$(i)$ for every process $i$ that it visits. Like the reading agent, the writing agent also traverses the network along a spanning tree. The spanning trees along which the reading and the writing agents traverse the network can be totally independent. The program for the writer is described in Fig 8:

When the writing agent returns home, the write operation is over. The home increments CLOK before the next traversal begins. The write operation does not require the cooperation of the individual nodes, except for the rejection of the messages that originated from nodes with a different value of the local clock. This requires that messages be stamped with the clock value of the sender.

4.4 Concurrent Reading and Writing

In the general case when a reading agent and a writing agent carry out their designated tasks in overlapped time, the writer may update the global state to different values during different traversals, and the reader, unaware of when and what the writer is writing, has to capture a consistent snapshot of the global state of the distributed system. There is no relationship between the speeds at which the writer and the reader move around the network. We will use the value of clock at the different nodes as the yardstick of progress. The value of clock is updated by the writing agent in all processes, including the home of the reader process. The reader and the writer agents traverse the network following distinct spanning trees, denoted in the algorithm by DFS$_R$ and DFS$_W$. On occasion, these trees may be identical, but there is no guarantee for it. Since the reader may be slow, the writing agent, in addition to updating the local state and the clock, will record the current state of the visited process into a set history for that process that could possibly be used by a slower reader. Each element in the history $h$ is a pair $(\text{clock}, \text{localstate})$, and we will designate the entry in the history of process $i$ corresponding to clock $j$ by $h^j(i)$. The saving of the current state becomes unnecessary when seq for the visited node is $-1$, since the state that will be read by the reader has already been saved by the process while updating seq to $-1$.

The following two observations are the cornerstones of the algorithm:

Lemma 4 \( \text{CLOK} (\text{writer}) = k \Rightarrow \forall i \text{ clock}(i) \in \{k - 1, k\}. \)

If the reading agent looks for copies of local states corresponding to clock $= k$ while the writer is still writing in round $k$, it is possible that the reader visits a node whose clock has not yet been updated from $k - 1$ to $k$. In this case, the reader will construct the snapshot from local states recorded at clock $k - 1$. All entries in the history corresponding to clock lower than $k - 1$ are of no use, and can be deleted by the reader. This leads to the following lemma:
Lemma 5 When the reader makes a traversal after the writer has started writing with a CLOCK \( k \), a consistent snapshot will be assembled from recordings of local states made at clock \( k \) only or \( k - 1 \) only.

Fig 9 (taken from [17]) shows the final program.

The 1-reader case can be easily extended to multiple readers, since readers do not interact – each process maintains a separate history. The extension to the multiple writer case is an open problem. The time complexity for a snapshot or a reset operation is determined by the time for one traversal. An issue of interest is the space complexity per process. Unfortunately, in the present version of the protocol, the space requirement can grow indefinitely when the writing operation is faster than the reading operation. The size of the briefcase for both the reader and the writer scales linearly with the size of the network. Bounding the space complexity when the writer is faster than the reader is an open problem. Also relevant is the issue of implementing various consistency models on the concurrent object.

5 Fault tolerance

When agents traverse an unknown network, they might get trapped in a host, known as a black hole. Once trapped, the agent is lost for all practical purposes, leaving no observable trace of destruction. A black hole need not always be a malicious host – for example, an undetectable crash failure of a host in an asynchronous network can make it a black hole.

An interesting aspect of fault-tolerance is finding out a black hole by sacrificing a minimum number of agents. The task is to unambiguously determine and report the location of the black hole, assuming there is only one black hole. The problem is called the black hole search (BHS) problem. More precisely, BHS is solved if at least one agent survives, and all surviving agents know the location of the black hole within a finite time. Black hole search is a non-trivial problem. In recent times, the problem has gained renewed significance as protecting an agent from "host attacks" has become a problem almost as pressing as protecting a host from an agent attack.

The problem of efficient black hole search has been extensively studied in many types of networks. The underlying assumption in most cases is that the network is totally asynchronous, i.e., while every edge traversal by a mobile agent takes a finite time, there is no upper bound on this time.

5.1 BHS in an Anonymous Ring

Model. Dobrev et al. [25] considered the BHS problem in the simplest symmetric topology: an anonymous ring \( R \), i.e., a loop network of identical nodes. Each node has two ports, labeled left and right. If this labeling is globally consistent, the ring is oriented, otherwise it is unoriented. Let \( 0, 1, \cdots, n-1 \) be the nodes of the ring in clockwise direction, and node 0 be the home base from where the agents start. Let \( A \) denote the set of anonymous mobile agents, and \( |A| = k \) denote the number of mobile agents. The asynchronous agents are assumed to have limited computing capabilities and bounded storage. They obey the same set of behavioral rules, i.e., the protocol. The bounded amount of storage in each node is called its whiteboard. Agents communicate by reading from and writing into the whiteboards, and access to a whiteboard is mutually exclusive.

When the anonymous agents start from the same node, they are termed as co-located agents. Otherwise, when they start from different nodes, they are called dispersed agents. The number of agents is the size of the fleet, and the total number of moves performed by the agents determines the cost for an algorithm. The following lemmas hold for the BHS problem:

Lemma 6 At least two agents are needed to locate the black hole.
Program for the writer while visiting process i

\{ The writer wants to reset the global state to W \}

agent variables \( \text{CLOCK}, W \);
process variables \( \text{clock}, s, h \);

if \((\text{clock}(i) < \text{CLOCK}) \rightarrow
    \text{if seq}(i) \neq -1 \rightarrow h(i) := h(i) \cup (\text{clock}(i), s(i)) \text{ fi;}
    s(i) := W.i; \text{clock}(i) := \text{CLOCK}
\) fi;
\text{NEXT} := DFS_W

Program for the reader while visiting process i

\{ The reader is trying to assemble a snapshot \( S \) \}

agent variables \( \text{SEQ}, \text{CLOCK}, S \);
process variables \( \text{seq}, \text{agent_seq}, \text{seq}, s, h \);

\( \forall j < \text{CLOCK} - 1 \text{ delete } h^j(i) \);

\{ Case 1 \} if \( \text{clock}(i) = \text{CLOCK} \rightarrow
    \text{if } \text{SEQ} = \text{agent_seq}(i) \oplus_3 1 \land \text{seq}(i) \neq -1 \rightarrow
        S.i.\text{CLOCK} := s(i); S.i.(\text{CLOCK} - 1) := h^{\text{CLOCK} - 1}(i)
    \boxcheck \text{SEQ} = \text{agent_seq}(i) \land \text{seq}(i) \neq -1 \rightarrow \text{skip}
    \boxcheck \text{seq}(i) = -1 \rightarrow S.i.\text{CLOCK} := h^{\text{CLOCK}}(i); S.i.(\text{CLOCK} - 1) := h^{\text{CLOCK} - 1}(i)
\) fi;

\boxcheck \{ Case 2 \} \text{clock}(i) = \text{CLOCK} - 1 \rightarrow
    \text{if } \text{SEQ} = \text{agent_seq}(i) \oplus_3 1 \land \text{seq}(i) \neq -1 \rightarrow
        S.i.(\text{CLOCK} - 1) := s(i); S.i.\text{CLOCK} := \text{null}
    \boxcheck \text{SEQ} = \text{agent_seq}(i) \land \text{seq}(i) \neq -1 \rightarrow \text{skip}
    \boxcheck \text{seq}(i) = -1 \rightarrow S.i.(\text{CLOCK} - 1) := h^{\text{CLOCK} - 1}(i); S.i.\text{CLOCK} := \text{null}
\) fi;

\boxcheck \{ Case 3 \} \text{clock}(i) > \text{CLOCK} \rightarrow
    \text{if } \text{SEQ} = \text{agent_seq}(i) \oplus_3 1 \rightarrow
        S.i.\text{CLOCK} := h^{\text{CLOCK}}(i); S.i.(\text{CLOCK} - 1) := h^{\text{CLOCK} - 1}(i)
    \boxcheck \text{SEQ} = \text{agent_seq}(i) \rightarrow \text{skip}
\) fi;
\text{seq}(i) := \text{SEQ}; \text{agent_seq}(i) := \text{SEQ};
\text{NEXT} := DFS_R

Figure 9: The concurrent reading and writing protocol
If there is only one agent, the BHS problem is unsolvable because the only agent will eventually disappear in the black hole.

**Lemma 7** It is impossible to find the black hole if the size of the ring is not known.

**Lemma 8** It is impossible to verify whether or not there is a black hole.

The presence of more than two agents does not reduce the number of moves. It can however be helpful in reducing the time spent by co-located agents to locate the black hole. The number of dispersed agents required to solve the problem depends on whether the ring is oriented or not. If the ring is oriented, then two anonymous dispersed agents are both necessary and sufficient. If the ring is unoriented, three anonymous dispersed agents are both necessary and sufficient [25].

**Cautious Walk**. *Cautious walk* is a basic tool in many BHS algorithms. At any time during BHS, the ports (corresponding to the incident links) of a node can be classified into three types:

- **Unexplored**: No agent has moved across this port.
- **Safe**: An agent arrived via this port.
- **Active**: An agent departed from this port, but no agent has arrived into it.

Both unexplored and active links are potentially dangerous, because they might lead an agent to the black hole. Only safe ports are guaranteed to be hazard-free. Cautious walk helps identify safe ports. It is defined by the following two rules:

**Rule 1.** When an agent moves from node \( u \) to \( v \) via an unexplored port (turning it into active), it immediately returns to \( u \) (making the port safe), and only then goes back to \( v \) to resume its execution.

**Rule 2.** No agent leaves via an active port.

**Theorem 9** In a ring with \( n \) nodes, regardless of the number of co-located agents, at least \((n - 1) \log(n - 1) + O(n)\) moves are needed for solving BHS.

**Sketch of solving BHS with two co-located agents.** We present the main idea behind the algorithm. It proceeds in phases. Let \( E_i \) and \( U_i \) denote the explored and unexplored nodes in phase \( i \), respectively. \( E_i \) and \( U_i \) partition the ring into two connected subgraphs, with the black hole located somewhere in \( U_i \). Divide the unexplored part of the ring between the two agents, assigning to each agent a region of almost equal size. Each agent starts the exploration of the assigned part. Because of the existence of the BH, only one of them will complete the exploration. When this happens, it will go through the explored part, until it reaches the last safe link visited by the other agent. It will then again partition the unexplored area in two parts of almost equal size, leave a message for the other agent (in case it is not in the BH), and go to explore the newly assigned area. If \(|U_{i+1}| = 1\), the surviving agent knows that the black hole is in the single unexplored node, and the algorithm terminates.

The two-agent algorithm is cost-optimal. There are algorithms for solving BHS in hypercubes and arbitrary graphs, which also follow the similar idea. We present a few results from [25]:

**Theorem 10** In the worst case, \( 2n - 4 \) time units are needed to find the black hole, regardless of the number of co-located distinct agents available.

**Theorem 11** The cost of locating the black hole in oriented rings with dispersed agents is at least \( \Omega(n \log n) \).
Theorem 12 If the agents have prior knowledge of the number of agents $k$, then the cost of locating the black hole in oriented rings is $\Omega(n \log(n - k))$

Theorem 13 In oriented rings, $k$ agents can locate the black hole in $O((n/\log n)/\log(k - 2))$ time, when $k$ is known.

5.2 BHS in Arbitrary Networks

Dobrev et al. studied topology independent generic solutions [26] for black hole search. The problem is clearly not solvable if the graph representing the network topology is not 2-connected. (BHS in trees requires a change of model that is discussed later). The cost and size for BHS algorithms are shown to be dependent on the a priori knowledge the agents have about the network, and on the consistency of the local labeling. The assumption is that all agents know $n$, the size of the network.

Model. Let $G = (V, E)$ be a simple 2-connected graph, $n = |V|$ be the size of $G$, $E(x)$ be the links incident on $x \in V$, $d(x) = |E(x)|$ denote the degree of $x$, and $\Delta$ denote the maximum degree in $G$. If $(x, y) \in E$ then $x$ and $y$ are neighbors of each other. The nodes are anonymous. At each node there is a distinct label called the port number associated to each of its incident links. Let $\lambda_z \in (x, z)$ denote the label associated at $x$ to the link $(x, z) \in E(x)$, and $\Lambda_z$ denote the overall injective mapping at $x$. The set $\lambda = \Lambda_z|x \in V$ of those mappings is called a labeling, and $(G, \lambda)$ is the resulting edge-labeled graph.

Let $P[x]$ denote the set of all paths with $x$ as a starting point, and let $P[x, y]$ denote the set of paths starting from $x$ and ending in $y$. Let $\Lambda$ be the extension of the labeling function $\lambda$ from edges to paths. A coding $c$ of a system $(G, \lambda)$ is a function such that: $\forall x, y, z \in V, \forall \pi \in P[x, y], \forall \pi_2 \in P[x, z]; c(\Lambda_x(\pi_1)) = c(\Lambda_x(\pi_2))$ iff $y = z$. For any two paths $\pi_1$ and $\pi_2$ from $x$ to $y$, $c(\Lambda_x(\pi_1)) = c(\Lambda_x(\pi_2))$. A decoding function $d$ for $c$ is such that $\forall x, y, z \in V$, such that $(x, y) \in E(x)$ and $\pi \in P[y, z]$, $d(\lambda_x(x, y), c(\Lambda_y(\pi))) = c(\lambda_x(x, y) \circ (\Lambda_y(\pi)))$, where $\circ$ is the concatenation operator. The couple $(c, d)$ is called a sense of direction for $(G, \lambda)$. If $(c, d)$ is known to the agents, the agents operate with sense of direction. Otherwise, the agents operate with topological ignorance. The agents have complete topological knowledge of $(G, \lambda)$ when the following information is available to all agents:

1. Knowledge of the labeled graph $(G, \lambda)$;
2. Correspondence between port labels and the link labels of $(G, \lambda)$;
3. Location of the home base in $(G, \lambda)$.

In [26], the authors proved the following results:

Theorem 14 With topological ignorance, there is an $n$ node graph $G$ with the highest degree $\Delta \leq n - 4$ such that any algorithm for locating the black hole in arbitrary networks needs at least $\Delta + 1$ agents in $G$. In addition, if $n - 4 < \Delta < n$ then any such algorithm needs at least $\Delta$ agents.

Theorem 15 With topological ignorance, there exists a graph $G$ such that any $\Delta + 1$ agent algorithm working on all 2-connected $n$-node networks of maximal degree at most $\Delta \geq 3$ needs $\Omega(n^2)$ moves to locate the black hole in $G$.

The authors provide an algorithm that correctly locates the black hole, in $O(n^2)$ moves using $\Delta + 1$ agents, where $\Delta$ is the highest degree of a node in the graph. They also showed the following:

Theorem 16 With topological ignorance, if $n - 3 \leq \Delta \leq n - 1$, $\Delta$ agents can locate the black hole with cost $O(n^2)$.
Theorem 17  In an arbitrary network with sense of direction, the black hole can be located by two agents with cost $O(n^2)$.

Theorem 18  The black hole can be located by two agents with full topological knowledge in arbitrary networks of vertex connectivity 2 with cost $O(n \log n)$, and this is optimal.

The lower bound of $\Omega(n \log n)$ in general networks does not hold for hypercubes and related networks. Dobrev et al. [27] provided a general strategy that allows two agents to locate the black hole with $O(n)$ moves in hypercubes, cube-connected cycles, star graphs, wrapped butterflies, chordal rings as well as in multidimensional meshes and tori of restricted diameter. Specifically they proved the following:

Theorem 19  Two agents can locate the black hole in $O(n)$ moves in all of the following topologies: tori and meshes of diameter $O(n/\log n)$, hypercubes, cube-connected cycles (CCC), wrapped butterflies and star graphs.

In another paper [29], Dobrev et al. showed that it is possible to considerably improve the bound on cost without increasing the size of the agents’ team. They presented a universal protocol that allows a team of two agents with a map of the network to locate a black hole with cost $O(n + d \log d)$, where $d$ denotes the diameter of the network. This means that, without losing its universality and without violating the worst-case $\Omega(n \log n)$ lower bound, their algorithm allows two agents to locate a black hole with $\Theta(n)$ cost in a very large class of, possibly unstructured networks, where $d = O(n/\log n)$.

5.3 BHS in Tree Networks

Model. The model for tree network is different. Obviously the assumption of 2-connectedness is no more valid for trees. Also the network is assumed to be partially synchronous instead of asynchronous. An upper bound on the time of traversing any edge by an agent can be established. Without loss of generality, we normalize this upper bound on edge traversal time to 1.

The partially synchronous scenario allows the use of time-out mechanism to locate the black hole in any graph, with only two agents. Agents proceed along edges of the tree. If they are at a safe node $v$, one agent goes to the adjacent node and returns, while the other agent waits at $v$. If after 2 units of time the first agent does not return, the other one survives and knows the location of the black hole. Otherwise, the adjacent node is safe, and both agents can move to it. This is a variant of the cautious walk. For any network, this version of BHS can be performed using only the edges of its spanning tree. Clearly, in many graphs, there are more efficient BHS schemes than those operating in a spanning tree of the graph.

Czyzowicz et al. [28] considered a tree $T$ rooted at node $s$, which is the starting node of both agents. It is assumed that $s$ is not a black hole. Agents have distinct labels. They can communicate only when they meet and not by leaving messages at nodes. There is at most one black hole in the network. Upon completion of the BHS there is at least one surviving agent and this agent either knows the location of the black hole, or knows that there is no black hole in the tree. The surviving agent(s) must return to the root $s$.

An edge of a tree is unknown, if no agent has moved yet along this edge (initial state of every edge). An edge is explored, if either the remaining agents know that there is no black hole incident to this edge, or they know which end of the edge is a black hole. In between meetings, an edge may be neither unknown nor explored when an unknown edge has just been traversed by an agent.

The explored territory at step $t$ of a BHS-scheme is the set of explored edges. At the beginning of a BHS-scheme, the explored territory is empty. A meeting occurs in node $v$ at step $t$ when the agents meet at node $v$ and exchange information which strictly increases the explored territory. Node $v$ is
**Approximation Algorithm** BHS on a Tree

explore(s) [The procedure explore(v) is described below for a general node. Initially v = s]

Procedure explore(v)
for every pair of unknown edges (v, x), (v, y) incident to v do
   split(x, y);
end for
if there is only one remaining unknown edge (v, z) incident to v then
   probe(z);
end if
if every edge is explored then
   repeat walk(s) until both agents are at s
else
   next := relocate(v);
   explore(next)
end if

Figure 10: Black hole search on a tree network.

called a *meeting point*. In any step of a BHS-scheme, an agent can traverse an edge or wait in a node. Also the two agents can meet. If at step t a meeting occurs, then the *explored territory* at step t is defined as the *explored territory* after the meeting. The sequence of steps of a BHS-scheme between two consecutive meetings is called a *phase*.

**Lemma 20** In a BHS-scheme, an unexplored edge cannot be traversed by both agents.

**Lemma 21** During a phase of a BHS-scheme an agent can traverse at most one unexplored edge.

**Lemma 22** At the end of each phase, the explored territory is increased by one or two edges.

**Lemma 23** Let v be a meeting point at step t in a BHS-scheme. Then at least one of the following holds: v = s or v is an endpoint of an edge that was already explored at step t − 1.

Czyzowicz et al. [28] provide an approximation algorithm (Fig 10) with ratio 5/3 for BHS in case of arbitrary trees. The time complexity of the algorithm is linear. It uses the following procedures:

**probe(v):** one agent traverses edge (p, u) (which is towards node v) and returns to node p to meet the other agent who waits. If they do not meet at step t + 2 then the black hole has been found.

**split(k, l):** One of the agents traverses the path from node m to node k and returns towards node p_k. The other traverses the path from node m to node l and returns towards node p_k. Let dist(l, k) denote the number of edges in the path from node k to node l. If they do not meet at step t + dist(l, k) then the black hole has been found.

**relocate(v):** This function takes as input the current node v where both agents reside and returns the new location of the two agents. If there is an unknown edge incident to a child of v then the agents go to that child. Otherwise, the two agents go to the parent of v.

The authors ask the open question whether there exists a polynomial time algorithm to construct a fastest BHS-scheme for an arbitrary tree. More generally, till now it is not known if the problem is polynomial for arbitrary graphs.

18
### Multiple Agents Rendezvous in a Ring in Spite of a Black Hole

The rendezvous problem requires all the agents to gather at the same node. Both nodes and agents, besides being anonymous, are also fully asynchronous. The assumption is that, there are $k$ asynchronous anonymous agents dispersed in a symmetric ring network of $n$ anonymous sites, one of which is a black hole. Clearly it is impossible for all agents to gather at a rendezvous point, since an adversary can direct some agents towards the black hole. So, Dobrev et al. [30] sought to determine how many agents can gather in the presence of a black hole.

The rendezvous problem $RV(p)$ consists in having at least $p \leq k$ agents gathering in the same site. There is no a priori restriction on which node will become the rendezvous point. Upon recognizing the gathering point, an agent terminally sets its variable to arrived. The algorithm terminates when at least $p$ agents set their arrived flag to true. A relaxed version of the rendezvous problem is the near-gathering problem $G(p, d)$ that aims at having at least $p$ agents within distance $d$ from one another. The summary of the results from the paper is given in Fig 11.

If $k$ is unknown then non-trivial rendezvous requires knowledge of the location of the back hole.

Here are some basic results about the rendezvous problem on a ring:

**Theorem 24** In an anonymous ring with a black hole

1. $RV(k)$ is unsolvable;
2. If the ring is unoriented, then $RV(k - 1)$ is unsolvable.

**Theorem 25** If $k$ is unknown, then non-trivial rendezvous requires locating the black hole.

**Theorem 26** If $n$ is not known, then the BH location is unsolvable.

**Theorem 27** Either $k$ or $n$ must be known for non-trivial rendezvous.

### BHS in Asynchronous Rings Using Tokens

Recently Dobrev, Kralovic, Santoro and Shi introduced a token model [31] for solving the BHS problem. A token is an atomic entity - agents communicate with one another and with the environment using these tokens. Each agent has a bounded number of tokens that can be carried and placed on a node, or removed from it. One or more tokens can be placed at the middle of a node, or on a port. All tokens are identical, and indistinguishable from one another.

There are no tokens placed in the network in the beginning, and each agent starts with some fixed number of tokens. The basic computational step of an agent, which it executes either when it arrives at a node or upon wake-up, is

1. (1) to examine the node.

---

3 $RV(p)$ is said to be non-trivial if $p$ is a non-constant function of $k$
(2) modify the tokens and either fall asleep or leave the node through either the left or right port.

The main results show that a team of two agents is sufficient to locate the black hole in a finite time even in this weaker coordination model, and this can be accomplished using only $O(n \log n)$ moves which is optimal, the same as with whiteboard model. To achieve this result, the agents need to use only $O(1)$ tokens each. Interestingly, although tokens are a weaker means of communication and coordination, their use does not negatively affect the solvability of the problem or lead to a degradation of performance. On the contrary, it turns out to be better in the sense, whereas the protocols using whiteboards assume at least $O(\log n)$ dedicated bits of storage at each node, the token algorithm uses only three tokens in total.

An open question is: whether it is possible to further reduce the number of tokens, and if so, then what will be the cost in such a scenario.

6 Stabilization using cooperating mobile agents

A distributed system occasionally gets perturbed due to transient failures that corrupt the memory of one or more processes, or due to environmental changes that include the joining of new process or the crash of an existing process. For example, in routing via spanning tree, if the tree is damaged due to a failure, then some packets will never make it to the destination. A stabilizing system [14] is expected to spontaneously recover from any perturbed configuration to a legal configuration by satisfying the following two properties [12]:

Convergence. Starting from an arbitrary initial configuration, the system converges to a legal configuration in a bounded number of steps.

Closure. If the system is a legal configuration, then it continues to do so unless a failure or a perturbation occurs.

Traditional stabilizing systems achieve stability using algorithms that use the message passing or the shared memory model of interprocess communication. In this section, we demonstrate how mobile agents can be used to stabilize a distributed system. The mobile agent creates an extra layer of computation that is superimposed on the underlying distributed computation, but does not interfere with it unless a failure occurs. The role of the mobile agent is comparable to that of a repairperson that roams the network, detects illegal configurations, and fixes it by appropriately updating the configuration. The individual processes are oblivious to the presence of the agent. We disregard any minor slowdown in the execution speed of a process due the sharing of the resources by visiting agents.

At any node, the arrival of an agent triggers the agent program whose execution is atomic. The agent program ends with the departure of the agent from that node, or with a waiting phase (in case the agent has to wait at that node for another agent to arrive), after which the execution of the application program at that node resumes. The computation at a node alternates between the agent program and the application program. At any node, the visit of a single agent can be represented by the following sequence of events. We denote an atomic event using $\langle \rangle$:

agent arrives, $\langle$ agent program executed $\rangle$, agent leaves

When a pair of agents $I$ and $J$ meet at a node $k$ to exchange data, the sequence of events will be as follows:

agent $I$ arrives, $\langle$ agent program of $I$ executed $\rangle$, agent $I$ waits at $k$

Following this, the application program at node $k$ resumes, and continues until the other agent $J$ arrives there. When $J$ arrives at node $k$, the following sequence of events take place:
agent $J$ arrives, (data exchange with $I$ occurs), agent $I$ and $J$ leave.

Then the application program at $k$ resumes once again.

Agent-based stabilization can be viewed as a stabilizing extension of a distributed system as proposed in [22]. While [22] emphasized the feasibility of designing stabilizing distributed systems, mobile agents have some interesting properties that make implementations straightforward. We first demonstrate the mechanism by presenting a stabilizing spanning tree construction [15] using a single mobile agent.

### 6.1 Stabilizing spanning tree construction using a single agent

To construct a spanning tree rooted at a given node of a graph $G = (V, E)$, we assume that the root is the home of the agent. Each node $i$ has a parent $p(i)$ chosen from its immediate neighborhood. In addition, each node $i$ has two other variables:

- $\text{child}(i) \equiv \{ j : p(j) = i \}$
- $\text{neighbor}(i) \equiv \{ j : (i, j) \in E \}$
- $\text{friend}(i) \equiv \{ j : j \in \text{N}(i) \wedge j \neq P(i) \wedge P(j) \neq i \}$

The program of the agent consists of three types of actions: (i) actions that update the local variables of the process that it is visiting, (ii) actions that modify its briefcase variables, and (iii) actions that determine the next process that it will visit. The individual processes are passive.

A key issue in agent-based solution is graph traversal. To distinguish between consecutive rounds of traversal, we introduce a briefcase variable $\text{SEQ} \in \{0, 1\}$ that keeps track of the most recent round of traversal. With every process $i$, define a boolean $f(i)$ that is set to the value of $\text{SEQ}$ whenever the process is visited by the agent. $\text{SEQ}$ is complemented by the root before the next traversal begins. Thus, the condition $f(i) \neq \text{SEQ}$ is meant to represent that the node has not been visited in the present round.

The agent program has three basic rules: DFS1, DFS2, DFS3 and is described$^4$ in Fig 12. The first rule DFS1 guides the agent to an unvisited child. The second rule DFS2 guarantees that when all children are visited, the agent returns to the parent node.

Unfortunately, this traversal may be affected when the DFS spanning tree is corrupted. For example, if the parent links form a cycle, then no tree edge will connect this cycle to the rest of the graph. Accordingly, using DFS1, the nodes in the cycle will be unreachable for the agent, and the traversal will remain incomplete. As another possibility, if the agent reaches one of these nodes contained in a cycle before the cycle is formed, then the agent is trapped, and cannot return to the root using DFS2.

To address the first problem, DFS3 will “force open” a path to the unreachable nodes. After reaching $k$, the agent will set $f(k)$ to true, and $p(k)$ to $i$, i.e., $i$ will adopt $k$ as a child). If the unreachable nodes form a cycle, then this rule will break it. This rule will also help restore the legal configuration, when the spanning tree is acyclic, but not a DFS tree.

To address the second problem, the agent will keep track of the number of nodes visited while returning to the root via the parent link. For this purpose, the briefcase of the agent will include a non-negative integer counter $C$. Whenever the agent moves from a parent to a child using DFS1 or DFS3, $C$ is reset to 0, and and when the agent moves from a node to its parent using DFS2, $C$ is incremented by 1. (Note: This will modify rules DFS1 and DFS2). When $C$ exceeds a predetermined value bigger than the size $N$ of the network, a new parent has to be chosen, and the counter has to be reset. This is the essence of DFS4. A proof appears in [15].

Both the time complexity and the message complexity for stabilization are $O(n^2)$. Once stabilized, the agent needs $2(n - 1)$ hops for subsequent traversals.

---

$^4$It disregards the details of how a process $i$ maintains its $\text{child}(i)$ and their flags.
Program for the agent while visiting node i

**agent variables** \(\textit{NEXT}, \textit{PRE}, \textit{SEQ}, \textit{C};\)

**process variables** \(f, \textit{child}, p, \textit{neighbor}, \textit{friend};\)

\[
\text{if } f(i) \neq \textit{SEQ} \rightarrow f(i) := \textit{SEQ} \text{ fi;}
\]

\[
\text{if } \textit{PRE} \in \textit{friend}(i) \rightarrow p(i) := \textit{PRE} \text{ fi;}
\]

\[
\text{if } \\
\quad \text{Visit an unvisited child} \\
\quad \text{(DFS1)} \exists j \in \textit{child}(i) : f(j) \neq \textit{SEQ} \rightarrow \textit{NEXT} := j; C := 0
\]

\[
\text{When all neighbors have been visited, return to the parent} \\
\text{(DFS2)} \forall j \in \textit{neighbor}(i) : f(j) = \textit{SEQ} \land (C < N \lor \textit{friend}(i) = \emptyset) \rightarrow \\
\quad \textit{NEXT} := p(i); C := C + 1
\]

\[
\text{Create a path to a node that is unreachable using DFS1} \\
\text{(DFS3)} \forall j \in \textit{child}(i) : f(j) = \textit{SEQ} \land \exists k \in \textit{neighbor}(i) : f(k) \neq \textit{SEQ} \rightarrow \\
\quad \textit{NEXT} := k; C := 0
\]

\[
\text{Break a possible cycle} \\
\text{(DFS4)} (C > N) \land \exists k \in \textit{friend}(i) \rightarrow \textit{NEXT} := k; p(i) := k; C := 0
\]

Program for the home process

**Executed when the agent visits home**

\[
\text{if } \exists j \in \textit{child}: f(j) \neq \textit{SEQ} \rightarrow \textit{NEXT} := j; C := 0
\]

\[
\text{\quad} \forall j \in \textit{neighbor}: f(j) = \textit{SEQ} \rightarrow \\
\quad \text{SEQ} := 1 - \text{SEQ}; \\
\quad C := 0; \textit{NEXT} := k : k \in \textit{child}
\]

fi

Figure 12: Spanning tree construction with a single agent.
6.2 Agent failure

If failures can hit the underlying distributed system, then they can hit the mobile agent too, resulting in its loss, or in the corruption of its state variables. This could do more damage than good and need to be addressed. To deal with agent failure, we first introduce a reliable agent. Divide the agent variables into two classes: privileged and non-privileged. Call an agent variable privileged, when it can be modified only by its home process - all other variables will be called non-privileged. Examples of privileged variables are SEQ, or the id assigned to an agent by its home process. Then, an agent will be called reliable, when it satisfies the following two criteria:

1. The agent completes its traversal of the network and returns home within a finite number of steps.
2. The values of all the privileged variables of the agent remain unchanged during the traversal.

An agent can be unreliable either due to the corruption of its privileged variables during a traversal, or due to routing problems. Note that, by simply being reliable, an agent cannot stabilize a distributed system. This leads to the adoption of a two-phased approach. In the first phase, we demonstrate how a reliable agent guarantees convergence and closure. In the second phase, we present methods by which unreliable agents eventually become reliable, and remain reliable thereafter, until the next failure occurs. This part will use some generic remedies, independent of the problem under consideration. The generic remedies are as follows:

**Loss of agent.** If the agent is killed, then the initiator discovers this using timeout and generates a new agent with a new sequence number. If the timeout is due to a delayed arrival of the original agent, then the original agent will eventually be killed\(^5\) by the initiator.

To avoid the risk of multiple agents with identical sequence numbers roaming in the system, the probabilistic technique of [19] can be used. It involves the use of a sequence number from a three-valued set \(\Sigma = \{0, 1, 2\}\). If the sequence number of the incoming agent matches with the sequence number of the previous outgoing agent, then the initiator randomly chooses the next sequence number from \(\Sigma\), otherwise the agent is killed.

**Corruption of the agent identifier.** An agent is recognized by its home using the agent’s id. If the id of the agent is corrupted, then the home process will not be able to recognize it, and the unreliable agent will roam the network forever. To prevent this, the supervisory program \(S\) of the agent counts the number of hops taken by the agent. As soon as this number exceeds a predefined limit \(c \cdot R\) (\(c\) is a large constant, and \(R\) is the roundtrip traversal time) the agent kills itself. The same strategy works, if due to routing anomalies the agent is unable to return home.

**Corruption of agent variables.** Here, the corruption of the non-privileged variables is not a matter of concern, because these are expected to be modified when the agent interacts with the underlying system. Our only concern is the possible corruption of privileged variables. To recover from such failures, we need to demonstrate that despite the corruption of the privileged variables, eventually the agent reaches the global state to which it was initialized by its home process. For each agent-stabilizing system, as a part of the correctness proof, we need to prove the following theorem:

**Theorem 28** An unreliable agent is eventually substituted by a reliable agent.

For the spanning tree generation algorithm using a single agent, all the generic fixes will hold. In addition, we will demonstrate how the system transparently handles an inadvertent corruption of the privileged variable SEQ.

---

\(^5\)This is a shift from the static to the dynamic model, but is necessary to keep the number of agents in control.
6.3 Spanning tree construction using multiple agents

We represent the agents by the upper case letters $I, J, K, \cdots$. Our model is a synchronous one, where in unit time, every agent takes a step. We assume that every agent visiting a process $i$ leaves its “footprint” by writing its own id to a local variable $f(i)$, which is a set of agent identifiers. Each agent can find out which other agents visited process $i$ by examining $f(i)$. The issue of bad data in $f(i)$ will be handled as follows: If an agent $J$ discovers the footprint of another agent $K$ that does not exist, then $J$ will delete the $K$’s footprint. Ordinarily all agents with a valid footprint are supposed to show up within a time period, otherwise they become fossils, and are flushed out.

We now employ multiple agents to generate a spanning tree (not necessarily DFS) of $g$, with the hope for reducing the message or time complexity. Our static model uses a fixed number $k$ of agents ($1 \leq k \leq n$). The proposed protocol is an adaptation of Chen-Yu-Huang protocol [13] for spanning tree generation. The home of the agent with the smallest id is designated the root of the spanning tree - we call it the root agent. The spanning tree generation has two layers: In the first layer, the agents work independently and continue to build disjoint subtrees of the spanning tree, until they meet other agents. In the second layer, the agents meet other agents to build appropriate bridges among the different subtrees – this results in a single spanning tree of the entire graph.

For the first layer, we will use protocol of Fig 12. Therefore, we will only elaborate on the second layer, where a pair of agents $K, L$ meet to make a decision about the bridge between them during an unplanned meeting at some process $x$. We will designate a bridge by the briefcase variable $BB$. When an agent $K$ that is yet to form a bridge meets another agent $L$ at a node $x$, it sets its briefcase variable $BB$ to $(L, x)$. Thereafter, node $x$ will have two parents $p_K$ and $p_L$ from the two subtrees generated by $K$ and $L$ (see node $j$ in Fig 13). Until a node $x$ is visited by an agent $K$, $p_K = \phi$.

![Figure 13: The spanning tree viewed as a graph with the nodes as subtrees and the edges as bridges.](image)

The maximum number of parents for any node is $\min(\delta, k)$ where $\delta$ is the degree of the node, and $k$ is the number of agents. By definition, the root agent does not have a bridge (we use $BB = \perp, \perp$ to represent this).

In addition to $BB$, we add another non-negative integer variable $Y(0 \leq Y \leq k)$ to the briefcase of every agent. By definition, $Y = 0$ for the root agent. Furthermore, during a meeting between two agents $K, L$, when $K$ sets up its bridge to $L, x$, it also sets $Y(K)$ to $Y(L) + 1$. Thus, $Y(K)$ denotes
Program for agent $K$ while meeting agent $L$ at node $i$

agent variables $BB$, $NEXT$, $PRE$;

process variables $p$ \{$\text{represents the parent of a node}$\};

initially $BB = ⊥, ⊥$;

\[\begin{split}
\text{do} & \quad BB(K) = L, i ∧ BB(L) = K, i ∧ K < L → BB(K) := ⊥, ⊥; \\
& \quad BB(K) = L, i ∧ BB(L) ≠ K, i ∧ K ≠ \text{root agent} ∧ Y(L) ≠ k → \\
& \quad \quad BB(K) := L, i; Y(K) := Y(L) + 1 \\
& \quad BB(K) = L, i ∧ BB(L) ≠ K, i ∧ Y(L) ≠ k ∧ Y(K) ≠ Y(L) + 1 → \\
& \quad \quad Y(K) := Y(L) + 1 \\
& \quad BB(K) ≠ L, i ∧ BB(L) = K, i ∧ p_{K}(i) ≠ PRE → p_{K}(i) := PRE \\
& \quad Y(K) = k ∧ Y(L) = k ∧ Y(K) ≠ k → Y(K) := k; \\
& \quad Y(K) = k ∧ BB(K) ≠ L, i ∧ Y(L) < k - 1 → \\
& \quad \quad BB(K) := L, i; Y(K) := Y(L) + 1 \\
& \quad BB(L) ≠ K, i ∧ p_{K}(i) ≠ φ → p_{K}(i) := φ \\
\text{od;}
\]

$NEXT := PRE$

Figure 14: Program for building a bridge between adjacent subtrees.

"how many subtrees away" the subtree of $K$ is from the root segment. In a consistent configuration, for every agent, $Y < k$. Therefore, if $Y = k$ for any agent, then the bridge for that subtree is invalidated.

Fig 14 describes the protocol for building a bridge between two subtrees. The home processes initialize each $BB$ to $⊥, ⊥$ once, but like other variables, these are also subject to corruption. The description of this protocol does not include the fossil removal actions.

Theorem 29 For a given graph, if each agent independently generates disjoint subtrees, then the protocol in Fig 14 stabilizes to a spanning tree that consists of all the tree edges of the individual subtrees.

Proof outline. As a consequence of the fossil removal mechanism, for every agent $K$, eventually $BB = ⊥, ⊥$ or $L, i$, where $i$ is a process visited by both $L$ and $K$. By definition, each subtree has exactly one bridge $BB$ linking it with another subtree. Draw a graph $g'$, in which the nodes are the subtrees (excluding the bridges) of $g$, and the edges are the bridges linking these subtrees. Using the arguments in [13], we can show that $g'$ will eventually be connected and acyclic. Therefore the set of edges (connecting a node with its parents) generated by the protocol of Fig 14 defines a spanning tree.

Note that any existing spanning tree configuration is closed under the actions of the protocol.

To estimate the complexities, assume that each subtree is of equal size $\frac{R}{s}$. Let $M(s_{K})$ be the number of messages required by a single agent $K$ to build a subtree of size $s_{K}$ starting from an arbitrary initial state. From [16], $M(s_{K}) = O(s_{K}^{2})$. Also, once the subtree is stabilized, the number of messages required to traverse the subtree is $2 \cdot (s_{K} - 1)$. Since we assume $s_{K} = \frac{n}{K}$, the number of messages needed to build the $k$ subtrees is $k \cdot M(\frac{R}{K})$. To estimate the number of messages needed to detect a cycle in the graph $g'$ using the condition $Y \geq k$, consider a cycle $s_{0}s_{1} \cdots s_{t}s_{0}$ ($t \leq k$) in $g'$, where each node is a subtree. To correctly compute $Y$, each agent has to read the value of $Y$ from the agent in its predecessor segment. This can take up to $1 + 2 + 3 + \cdots + (t - 1) = \frac{t(t - 1)}{2}$ traversals of the subtrees. Since the maximum value of $t$ is $k$, for correctly detecting cycles in $g'$, at most $\frac{k(k - 1)}{2} \cdot \frac{2}{K}$ will be required. Also, each time a cycle is broken, the number of disjoint subtrees
in $g'$ is reduced by one (see [13]), so this step can be repeated no more than $(k - 1)$ times. Therefore the maximum number of messages needed for the construction of a spanning tree using a set of cooperating reliable agents will not exceed

$$k \cdot O\left(\frac{n^2}{k^2}\right) + (k \cdot \frac{k(k-1)}{2} \cdot \frac{2n}{k}) = O\left(\frac{n^2}{k} + n \cdot k^2\right)$$

To estimate the worst-case message complexity, we also need to take into account the overhead of fossil removal. This is determined by the number of hops taken by the agents to traverse the subtrees of size $\frac{n}{k}$, which is $O\left(\left(\frac{n}{k}\right) \cdot k\right) = O(n)$. Note that this does not increase the order of the message complexity any further. The interesting result, at least with this particular protocol is that as the number of agents increases, the message complexity first decreases, and then increases. The minimum message complexity is $O(n^\ast)$ when $k = O(n^\ast)$.

To estimate the time complexity, assume that each of the $k$ agents simultaneously builds subtrees of size $\frac{n}{k}$ in time $O\left(\frac{n^2}{k}\right)$. The time required by the $k$ agents to correctly establish their $Y$ values is $k \cdot O\left(\frac{n^2}{k}\right) = O(n)$. At this time, the condition $Y \geq k$ can be correctly detected. The resulting actions reduce the number of disjoint subtrees by 1, so these action can be repeated at most $(k - 1)$ times. The time complexity is thus $O\left(\frac{n^2}{k} + n \cdot k\right)$. The overhead of fossil removal (which is $O\left(\frac{n}{k}\right)$) does not increase the time complexity any further. Therefore, the smallest value of the time complexity is $O(n^\ast)$ when $k = O(n^\ast)$.

7 Conclusion

Of the various possible roles that multiple mobile agents can play in a distributed system, this article picks four specific applications, illustrates how they work, and summarize important results and open problems. For the proofs of these results, we encourage the readers to read the original articles.

Some of these problems can be solved using a single agent too. But a few problems (like the BHS problem) cannot be solved by a single agent, and explicitly needs multiple agents. The multiplicity of agents, on one hand, accelerates certain applications, but on the other hand, synchronization becomes a tricky issue. Peleg [4] highlights several open problems related to coordination of multiple autonomous mobile robots (also known as robot swarms) for distributed computing application.

References


