Brief Announcement:
Self-Stabilizing Resource Discovery Algorithm*

Seda Davtyan
Computer Science & Engineering
University of Connecticut
sedajengr.uconn.edu

Kishori M. Konwar
Immunology and Microbiology
University of British Columbia
kishori@interchange.ubc.ca

Alexander A. Shvartsman
Computer Science & Engineering
University of Connecticut
aas@cse.uconn.edu

ABSTRACT
Distributed cooperative computing in networks involves marshaling collections of network nodes possessing the necessary computational resources. Before the willing nodes can act in a concerted way they must first discover one another. This is the general setting of the Resource Discovery Problem (RDP). This paper presents a self-stabilizing algorithm that solves RDP in a deterministic synchronous setting. The solution approach is formulated in terms of evolving knowledge graphs, where vertices represent the participating network nodes, and edges represent one node’s knowledge about another. Ideally, the diameter of such a graph is one, i.e., each node knows all others. The algorithm works in rounds as it evolves the knowledge graph with the goal of reducing its diameter. This is accomplished by nodes sharing their knowledge through gossip messages. We prove that the algorithm is self-stabilizing, i.e., it tolerates arbitrary perturbations in the nodes’ local states and is guaranteed to solve the problem once such failures subside. The algorithm has stabilization time of $O(D)$, and it takes at most $4D + 4$ complete rounds to stabilize, where $D$ is the diameter of the initial knowledge graph, and the corresponding message complexity is $O(|V| \cdot D)$, where $V$ is the set of participating nodes.

Categories and Subject Descriptors: F.2.0 [Theory of Computation]: ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY — General

Keywords: Resource Discovery; Self-Stabilization; Distributed Cooperation; Fault-Tolerance

1. INTRODUCTION
A large collection of networked computers may need to cooperate in implementing a distributed system, for example, to provide a shared data service, or to perform a set of tasks. The necessary first step in such settings is to discover the relevant computational resources in the network. This step is formalized as the Resource Discovery Problem, where each willing resource must find all other resources willing to collaborate. This problem was introduced by Harchol-Balter, Leighton, and Lewin [5]. Kutten, Peleg, and Vishkin [6] gave an efficient deterministic algorithm for the problem. However, it does not have strong fault-tolerance properties and does not deal with dynamic situations, thus its guarantees do not hold in the presence of failures. Additionally, it assumes that certain knowledge of the nodes is non-decreasing.

Our goal is to design algorithms that can deal with transient failures, and in particular we are interested in self-stabilizing solutions, cf. [3]. Here the algorithm must be able to bring a system into a legitimate state in spite of arbitrary state corruptions, and once failures subside, such a legitimate state is reached in a finite time. For additional details we refer the reader to [2].

2. THE PROBLEM AND SYSTEM MODEL
We consider the Resource Discovery Problem (RDP) in deterministic synchronous settings. Let there be a universe of processes, with unique identifiers from a well-ordered set $U$. Let $V \subseteq U$ be the set of processes chosen by the environment to participate in the computation. We let $n_0$ stand for $\min\{v : v \in V\}$. The set $V$, its cardinality, and $v_0$ are unknown to the processes, but each process $v$ in $V$ is aware of one other process in $V$: each $v$ has a constant $nk = V$, where $v \neq nk$, representing the knowledge of $v$ of some other process (a neighbor). This induces a directed graph.

**Definition 1.** Given the set $V$ and $nb$, for all $v \in V$, we define the connectivity graph as the directed graph $G = (V, E)$, where $E = \{(u, v) : nb_v = v\}$.

We assume that the connectivity graph is at least weakly-connected, representing the setting where each process has the knowledge of at least one other process (as in the original formulation in [5]). We assume that each process $v$ has three local variables, $prt_v \in V$, $C_v \in 2^V$ and $K_v$, where $prt_v = u$ means that $v$ considers $u$ to be its parent, $u \in C_v$ means that $v$ considers $u$ to be its child, with $C_v$ being the set of all children of $v$, and finally $u \in K_v$ means that $v$ knows $u$. We now define the Resource Discovery Problem.

**Definition 2.** Given the weakly-connected graph $G$, the Resource Discovery Problem (RDP) is to establish and maintain the following invariant on configurations:

$(\exists v \in V : (C_v = V) \land (\forall u \in V : prt_u = v)) \land (\forall u \in V : K_u = V)$, that is, (1) there exists a node $v \in V$ such that $C_v = V$, and (2) for every node $u \in V$ we have $prt_u = v$, and (3) for every node $u \in V$ we have $K_u = V$. 

*This work is supported in part by the NSF award 1017232.
For convenience we let $G^n$ be the undirected graph induced by $G$, called the initial knowledge graph. Let $D$ be the diameter of $G^n$ and $\text{dist}(u,v)$ be the length of the shortest path from node $u$ to $v$ in $G^n$.

The nodes communicate using point-to-point messages. Sending (multicasting) messages requires that the sending node has the identifiers of the destination nodes (arbitrary broadcast is not allowed). The communication is synchronous in the sense that there is a known upper bound $d$ on message delays. If a node expects a message from another node and the message is sent, then it is delivered within $d$ time units. Nodes have access to synchronous timers that can be used to implement message time-outs. Local computation takes negligible time relative to $d$. We do not assume that all nodes begin participating in the computation simultaneously; instead we allow the nodes to join the computation at arbitrary times. At a high level, the computation is structured in terms of synchronous rounds, however the activities within each round are not synchronized across the nodes.

The nodes are subject to arbitrary perturbations to their local (volatile) states and arbitrary crash and restart events that occur in matched pairs. The static code of each node, its constants, and the clock are incorruptible. All other variables are subject to corruption. Here a corrupted variable may contain a value that is syntactically indistinguishable from a valid value. This is in contrast with some works in self-stabilization, where failures cause erasures of variable values, making such failures easily detectable, cf. [4]. Other works, e.g., [7], assume that any node identifier must represent an actual node in the system. Finally, we also allow the adversary to corrupt messages in transit.

We denote a transition from configuration $\sigma_i$ to $\sigma_{i+1}$ by $\sigma_i \xrightarrow{r} \sigma_{i+1}$ and we let $\sigma \xrightarrow{r} \sigma'$ stand for the fact that $\sigma'$ can be reached from $\sigma$ by zero or more transitions.

**Self-stabilization** is the ability of a system to recover from transient failures following their cessation. The impact of a failure is that the transition from configuration $\sigma$ to configuration $\sigma'$ may not obey the transition function $\tau$, that is, a failure may cause $\sigma' \neq \tau(\sigma)$. Thus we assume that the local state of any node can be corrupted, and in particular, that a system can start in any configuration. In designing solutions resilient to transient failures we will use self-stabilization techniques, formalized in terms of closure and convergence properties (cf. [1]).

**Definition 3. (Self-stabilization)** Let problem $P$ be to establish and maintain invariant $\psi()$, given as a predicate on configurations. System $S = (\Sigma, A, \tau)$ is a self-stabilizing solution for problem $P$, if the following two conditions hold:

**Closure:** $\forall \sigma \in \Sigma \forall a \in A : \psi(\sigma) \implies \psi(\tau(\sigma,a))$, i.e., $\tau$ maintains the invariant.

**Convergence:** $\forall \sigma \in \Sigma : \exists \sigma' \in \Sigma : \sigma \xrightarrow{r} \sigma' \land \psi(\sigma')$, i.e., $\psi()$ can be established in the absence of failures.

**Measures of efficiency.** We assess the efficiency of the algorithm in terms of stabilization time and stabilization message complexity. The stabilization time is measured in terms of the worst case number of rounds following the cessation of perturbations needed to establish the resource discovery invariant. The stabilization message complexity is measured in terms of the worst case number of point-to-point messages sent among the participants to establish the resource discovery invariant following the cessation of perturbations.

**Data-types:**
- $U$, the set of node identifiers
- $M$, the set of messages

**Constants:**
- $nb : U$
- $t : \text{real} > 0$

**Derived Constants:**
- $\bar{N} = \{i\} \cup \{nb\}$

**Signature:**
- Input:
  - msend$(m, u)_i$, $m \in M$, $u \in U$
  - join
  - perturb
- Output:
  - msend$(m, I)_i$, $m \in M$, $I \subseteq U$

**State:**
- $active : \text{bool}$
- $phase : \{\text{gossip, confirm}\}$
- $clock : \text{real}$
- $prt : U$
- $pp : U$
- $do-msend : \text{bool}$

**Figure 1:** Signature and state of $RD_i$ at node $i$ in $V$

### 3. ALGORITHM RDS

The algorithm has an iterative structure, where each iteration consists of two synchronous rounds. The first round is referred to as the *gossip* phase and the second round as the *confirm* phase. In both phases nodes propagate information to other nodes, while in the *confirm* phase the nodes additionally validate the identities of the nodes to whom the information was propagated in the *gossip* phase.

The behavior of each node $i \in V$ is specified as a timed I/O automaton, called $RD_i$. The specification is given in Figure 1 that defines constants, signature, and state variables, and Figure 2 that defines the transitions and the trajectory. The full system, called $RDS$, is the composition of automata $RD_i$ for $i \in V$, the multicast implementation, and the $Channel_{i,j}$ automata for $i, j \in V$ (not specified here).

The main variables are $active_i$, $C_i$, $K_i$, $Nbrs_i$, and $prt_i$. Boolean $active_i$ indicates whether node $i$ is active or not, set $C_i$ contains the children of node $i$, set $K_i$ contains its siblings in the evolving knowledge graph, set $Nbrs_i$ contains the identifiers of the nodes that $i$ considers to be neighbors in $G^n$, and $prt_i$ is the identifier of the node that $i$ considers to be its parent. The remaining variables are used for control: $phase_i$, $pp_i$, $New_C_i$, $do-msend_i$, $R_i$, and $Dest_i$.

The environment may activate node $i$ by using input action $join_i$, and it may disable and/or corrupt the state of node $i$ by means of input action $perturb_i$, where HAVOC assigns arbitrary values to the state variables, modeling a transient failure. If HAVOC sets $active_i$ to false, the action models a crash of the node. Internal action $restart_i$ is always enabled, modeling the assumption that each node $i \in V$ is eventually active. Nodes gossip by sending and receiving messages through actions $msend_i$ and $msend_{ij}$.

Variable $clock_i$ represents the time of the synchronous system. Recall that failures cannot change the synchronous nature of the system, and thus this is the only variable that is not affected by transient failures. The constant $t$ is used to control the duration of rounds ($t$ is readily obtained from the structure of the algorithm and from the knowledge of the worst case message delivery delay $d$). The trajectory specification says that time “stops” when $clock_i \% t = 0$ for
an active node. The value of clock is used to determine whether an active node is in the gossip or confirm phase.

A round ends with either action end-round, or action reset. Action end-round, is enabled every t time units (specified by clock[t = 0 in the code) at the conclusion of each round if the node’s state suggests that its parent is active (this does not mean that perturbations did not occur). Action reset, is enabled every 2t time units (when clock % 2t = 0) and the parent does not respond during the iteration. In this case the node gives up, resets its state and starts anew.

4. ALGORITHM ANALYSIS

Our analysis shows that algorithm RDS satisfies the Closure and Convergence properties of Definition 3. In the analysis we let A denote the set of all actions of the algorithm, excluding actions join and perturb. 

THEOREM 1. (Closure) Consider any execution prefix of RDS consisting of complete iterations, where \( \sigma \) is the final configuration. If \( \sigma \) is legitimate, then any extension of the execution by up to one complete iteration using only the actions from A results in \( \sigma \xrightarrow{t} \sigma' \), where \( \sigma' \) is a legitimate configuration.

We prove the following convergence property.

THEOREM 2. (Convergence) Consider an execution prefix of RDS that ends with configuration \( \sigma \). Any fair extension of the execution of a sufficient length that uses only the actions from A reaches a configuration \( \sigma_i \) in at most \( 2D + 2 \) complete iterations, such that \( \sigma_i \) is a legitimate configuration.

Finally we reason about the efficiency of the algorithm.

THEOREM 3. Any execution prefix of RDS ending in an arbitrary configuration can be infinitely extended to solve the resource discovery problem. The stabilization time of the algorithm is \( O(D) \), taking at most \( 4D + 4 \) complete rounds to stabilize. The stabilization message complexity is \( O(|V| \cdot D) \).

Recall that an important goal of the algorithm is to manage the overall communication complexity in the presence of perturbations. To limit the number of messages sent to bogus destinations, the algorithm refreshes the states of the nodes in each iteration. The convergence of the algorithm in \( O(D) \) rounds is largely due the fact that it does not aggregate knowledge across multiple iterations.

Lastly we note that if the initial knowledge graph is not connected, or if permanent crashes disconnect the graph, then our algorithm solves the problem for each (weakly) connected component. Our follow up work will focus on improving efficiency and stronger adversarial behaviors.

5. REFERENCES


