A distributed algorithm for the graph center problem

Linlin Song
Mathematics Department, University of York
Supervisor: Keith Briggs

Complexity research group, BT Exact, Martlesham
mailto:ls150@york.ac.uk, mailto:keith.briggs@bt.com

August 29, 2003

Contents

List of Figures

Abstract

In this dissertation I am going to introduce a new asynchronous distributed algorithm for the graph center problem, and a simulation program based on this algorithm. This simulates how to find a center node in a distributed multihop network, so that we solve a location problem. The algorithm is based on three subalgorithms (test connectivity, all-pair shortest path, and leader election); each operates in a different layer. The simulation program automatically generates a random connected network for testing. All the nodes run the same code. The cost of communication through a link is the number of hops it contains. This program can be slightly changed to find the median of a graph, which is solution of another location problem. This program will also provide all-pair shortest path information by providing the first neighbor identification on the corresponding shortest path. The full program code and introduction of usage is provided. This report contains reference material on the theoretical part for ease of reading.

Chapter 1

Introduction

1.1 What is a distributed system?

We use the term *distributed system* to mean an interconnected collection of autonomous computers, processes, or processors. They are all referred to as the nodes of the distributed system. To be qualified as 'autonomous', the nodes must at least be equipped with their own private control. To be qualified as 'interconnected', the nodes must be able to exchange information.

As software or process can play the role of nodes of a system, the definition includes software systems built as a collection of communicating processes, even when running on a single hardware installation. In most cases, however, a distributed system will at least contain several processors, interconnected by communication hardware.

More restrictive definitions of distributed systems are also found in the literature. Tanenbaum [Tan88], for example, considers a system to be distributed only if the existence of autonomous nodes is *transparent* to users of the system. A system distributed in this sense behaves like a virtual, standalone computer system, but the implementation of this transparency requires the development of intricate distributed control algorithm.

1.2 Complex networks

Complex networks occur in diverse areas from metabolic and gene regulation networks in each cell, food web in ecology, transportation networks, economic interactions and the organization of the internet. A network is conveniently modelled as a graph G which consists of a set of vertices and a set of edges which we regard as pairs of distinct vertices. various of network and graph models are discussed next.

1. Transport network: To simulate the network of interaction between objects in the simulation, where these objects are not only travellers, but also traffic signals, traffic management centers, etc. For fast large scale simulations, one employs distributed computers, and mapping these interactions on the computational system is critical for high computing performance. Travellers and other and other objects in a transportation system interact. For example, congestion is formed by travellers being in each other's way; ride sharing necessitates travellers to meet at a common pick-up location; adaptive traffic lights react to traffic conditions; etc.There are also increasingly networked services which are both sparse and non-local for example electronic route guidance systems where only very few travellers communicate with a center.

2. Economics network: Almost any serious consideration of economics organization leads to the conclusion that network structures both within and between organizations are important. The idea that networks of relations at various levels have an important effect on economic activity is familiar in sociology. we consider the interaction structure as being modelled by a graph where the agents are the nodes and two nodes are linked by an edge if the corresponding agents interact. We assume that each agent is influenced only by a limited (finite) number of other agents who are within a certain distance of him. Such individuals are usually referred to as the agent's 'neighbors'. As agents' interactions are limited to a set of neighbors, changes will not affect all agents simultaneously but rather diffuse across the economy. The way in which such diffusion will occur, and the speed with which it happens, will depend crucially on the nature

of the neighborhood structure. It is the connectivity of the graph of relations that will be essential here.

1.3 Distributed sensor network

A distributed sensor network is a network which is composed of sensors distributed in the environment. Sensors include: cameras as vision sensors, microphones as audio sensors, ultrasonic sensors, infra-red sensors, temperature sensors, humidity sensors, force sensors, pressure sensors, vibration sensors.

Of course, they are not limited to them. One particular example in wireless mobile communication is *Mobile ad hoc networks* a mobile ad hoc network is a network wherein a pair of nodes communicates by sending messages either over a direct wireless link, or over a sequence of wireless links including one or more intermediate nodes. Only pairs of nodes that lie within one another's transmission radius can directly communicate with each other. Wireless link 'failures' occur when previously communicating nodes move such that they are no longer within transmission range of each other. Likewise, wireless link 'formation' occurs when nodes that were too far separated to communicate move such that they are within transmission range of each other. Distributed sensors can cover a large space even if the sensing range of each sensor is limited. That is, by integrating many sensors information, we can acquire diverse and precise information of the environment where those sensors cover. Figures 1.1 and 1.2 below illustrate an application space figure 1.1and presumable real applications figure 1.2 of the distributed sensor network, respectively.

1.4 Geometric centrality

Geometric notions of centrality are closely linked to facility location problems. Suppose, we give a graph G representing, say, a traffic network. We may then ask questions such as the following:

(a) What is the optimal location of a hospital such that the worst case response time of an ambulance is minimal?

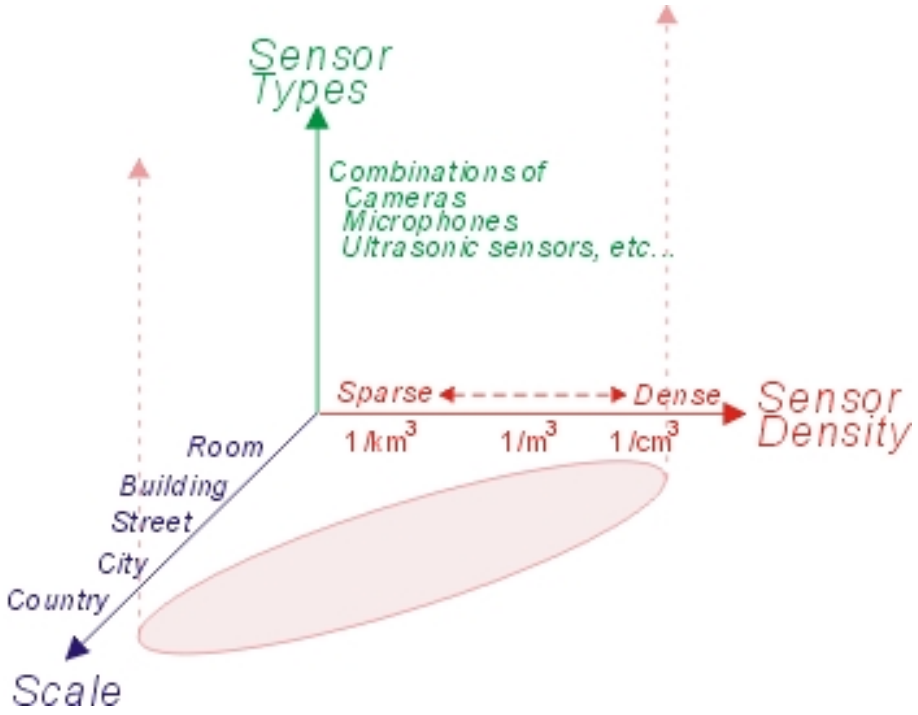

Figure 1.1: Application space of a distributed sensor network

(b) What is the optimal location of a shopping mall so that the average driving time to the mall is minimal?

These two classical facility local problems can be recast as optimization problems based on the distance matrix $D = (d(x, y))$ of G. Their solution define two different notions of 'central' vertices.

The *eccentricity* of a vertex x in G and the radius $\rho(G)$, respectively, are defined as $e(x) = \max d(x, y), y \in V$ and $\rho(G) = \min e(x), x \in V$. The center of G is the set $C(G) = \{ x \in V | e(x) = \rho(G) \}$. $C(G)$ is the solution of the 'emergency facility local problem' (A) which is always contained in a single block of G.

The *status* $d(x)$ of a vertex and the status $\sigma(G)$ of the graph G, respectively, are defined as $d(x) = \sum_{y \in V} d(x, y)$ and $\sigma(G) = \min d(x)$.

The *median* is the solution of the 'service facility location problem' (B). The *median* of G is the set

$$M(G) = \{ x \in V | d(x) = \sigma(G) \}.$$

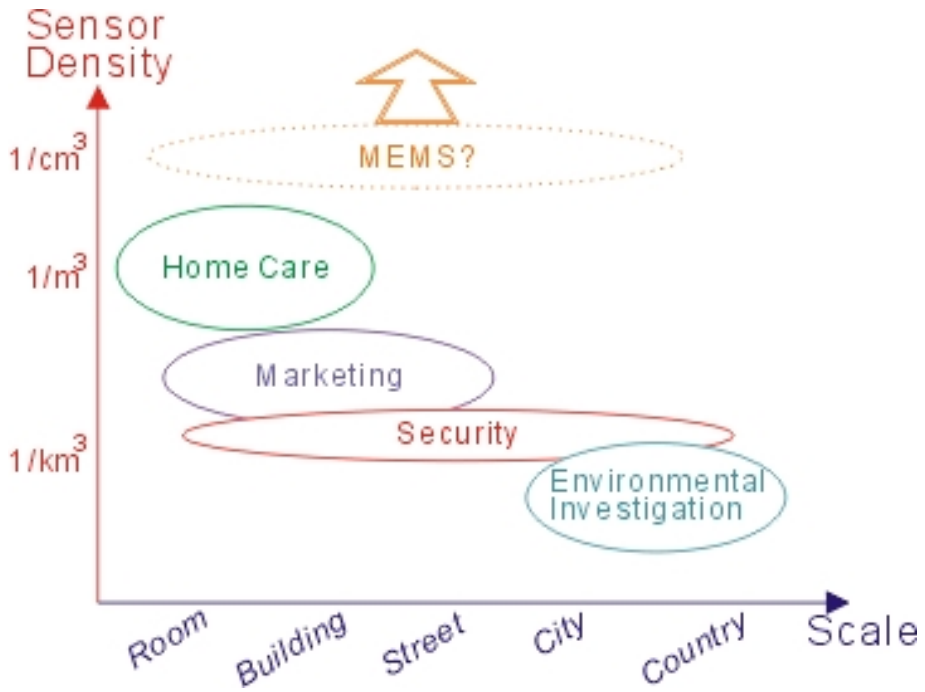

Figure 1.2: Presumable real applications

There are a few center algorithms in the literature, but they all have their own limitations, for example limit to particular network topology, complete graph, tree, ring, or is designed to be synchronous. In this report, I present a new asynchronous center algorithms for arbitrary network, this algorithm is highly adaptive, it could be used to solve the median problem and also can slightly changed to solve other more specific optimization problems.

Chapter 2

The model of computation

2.1 Distributed algorithms

The previous chapter have given reasons for use of distributed computer systems and explained the nature of these systems; the need to program these systems arises as a consequence. The programming of distributed systems must be based on the use of correct, flexible, and efficient algorithms.Distributed and centralized systems differ in a number of essential respects:

(1) Lack of knowledge of global state. In a centralized algorithm control decisions can be made based upon an observation of the state of the system. Even though the entire state usually cannot be accessed one by one.

(2) Lack of global time-frame. The event constitute the execution of a centralized algorithm are totally ordered in a natural way by their temporal occurrence; for each pair of events, one occurs earlier constituting the execution of a distributed algorithm is not total; for some pairs of events there may be a reason for deciding that one occurs before the other, but for other pairs it is the case that neither of the events occurs before the other.

(3) Non-determinism. A centralized program may describe the computation as it unrolls from a certain input unambiguously; given the

program and the input, only a single computation is possible. In contrast, due to possible differences in execution speed of the system components.

2.2 The model

2.2.1 Introduction

Distributed applications are pervading many aspects of everyday life. Booking reservations, banking, electronic and point-of-sale commerce are noticeable examples of such applications . Those applications are built on top of distributed systems. We present a distributed algorithm by the connected directed graph $G_T = (N_T, D_T)$, where the node set N_T is a set of tasks and the set of directed edges D_T is a set of unidirectional communication channels.

The communication in a distributed system relies on the message passing. A node is reactive (or message-driven) entity, in the sense that normally it only performs computation (including the sending of messages to other nodes) as a response to the receipt of a message from another node.

These are some points of the asynchronous distributed algorithm model. More details will be added later in this chapter.

1. Network of identical nodes, each with message queue. This means that each node of the network has a unique name. The message queue may contains more than one buffer depends on how many layered a node contains. The reason to be a queue is to ensure the FIFO order of message processing.

2. Each knows only its neighbours. This assumes that there exists a lower layer to provide nodes with neighbours list. Generally this list contains the identification or address of the neighbours.

3. $\exists$ a finite set of pre-specified messages. This means, since the communication in a distributed system relies on the message passing or interaction of nodes. Different message corresponds with different state of the system.

4. Each performs the same subalgorithm. This means each node has the ability of computation. An distributed algorithm is designed to deal with all the possible states the system may have, and this apply to all the node in the same system.

5. Links is reliable and links between each pair of nodes deliver interprocessor messages in the FIFO order. This is not necessary in all distributed algorithm, but this is needed for those we are going to discuss in this report.

6. Indefinite delay before reply to message. This is made well suited to the asynchronous situation.

7. Each runs asynchronously with respect to neighbours. This means that it is impossible to define an upper bound on process scheduling delays and on message transfer delays. This is due to the fact that neither the input load from users nor the precise load of the underlying network can be accurately predicted. This means that whatever is the value used by a process to set a timer, this value cannot be trusted by the process when it has to take a system-wide consistent decision.

2.2.2 Full asynchronism and full synchronism:

when we investigate the nature of the computation carried out by G's nodes with respect to their timing characteristics. This investigation will enable us to complete the model of computations given by G with the addition of its timing properties.

The first model we introduce is the fully asynchronous (or simply asynchronous model), which is characterized by the following two properties.

1. Full asynchronism

   (1) Each node is driven by its own, local, independent time basis, referred to as its local clock.

   (2) The delay that a message suffers to be delivered between neighbors is finite but unpredictable.

2. Full synchronism

   (1) All nodes are driven by global time basis, referred to as the global clock, which generates time intervals (or simply intervals) of fixed, nonzero duration.

   (2) The delay that a message suffers to be delivered between neighbors is nonzero and strictly less than the duration of an interval of the global clock.

   The complete asynchronism assumed in this case makes it very realistic from the standpoint of somehow reflecting some of the characteristics of the distributed systems.

2.2.3 The model

There are various of distributed computing models in the literature, for example, [Bar96a], [BT96], [Lyn97], [RS99], I like the one of Gerard Tel [Ger91]. Next I am going to describe Tel's model.

A distributed computation is usually considered as a collection of discrete events, each event being an atomic change in the configuration (the state of the entire system). This notion is captured in the definition of *transition systems*, leading to the notion of reachable configurations and a constructive definition of the set of executions induced by an algorithm. What makes a system 'distributed' is that each transition is only influenced by, or only influences, part of the configuration, basically the (local) state of a single process. (Or, the local states of the subset of interacting processes.)

1. Transition Systems and Algorithms. A system whose state changes in discrete steps (transitions or events) can usually be conveniently described by the notion of a *transition system*. In the study of distributed algorithms this applies to the distributed system as a whole, as well as to the individual processes that cooperate in the algorithm. Therefore, transition systems are an important concept in the study of distributed algorithms.

   Processes in a distributed system communicate either by accessing shared variables or by message passing. We shall take a more re-

strictive point of view and consider only distributed systems where communication is by means of exchanging messages.

Messages in distributed systems can be passed either synchronously or asynchronously. Our main emphasis is on algorithms for systems where messages are passed asynchronously. For many purposes synchronous message passing can be regarded as special cans of asynchronous passing, as was demonstrated by Charron-Bost *et al*. [CBMT92].

2. Transition Systems. A transition system consists of the set of all possible states of the system. The transitions ('moves') the system can make in this set, and a subset of states in which the system is allowed to start. To avoid confusion between the states of a single process and the states of the entire algorithm (the 'global states'). The latter will from now on be called configuration.

Definition 2.1 *A transition system is a triple $S = (\mathcal{C} \rightarrow \mathcal{I})$, where $\mathcal{C}$ is a set of configuration, $\rightarrow$ is a binary transition relation on $\mathcal{C}$, and $\mathcal{I}$ is a subset of $\mathcal{C}$ of initial configurations.*

A transition relation is a subset of $\mathcal{C} \times \mathcal{C}$. Instead of $(\gamma, \delta) \in \rightarrow$ the more convenient notation $\gamma \rightarrow \delta$ is used.

Definition 2.2 *Let $S = (\mathcal{C}, \mathcal{I})$ be a transition system. An execution of S is a maximal sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$, where $\gamma_0 \in \mathcal{I}$, and for all $i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$.*

A terminal configuration is a configuration γ for which there is no such that δ. Note that a sequence $E = (\gamma_0, \gamma_1, \gamma_2, ...)$ with $i \geq 0, \gamma_i \rightarrow \gamma_{i+1}$ for all i, is maximal if it is either infinite or ends in a terminal configuration.

Definition 2.3 *Configuration δ is reachable from γ, notation ,$\gamma \rightarrow \delta$, if there exists a sequence $\gamma = \gamma_0, \gamma_1, \gamma_2, ...\gamma_k\delta$ with $\gamma_i \rightsquigarrow \gamma_{i+1}$ for all $0 \leq i < k$. Configuration δ is reachable if it is reachable from an initial configuration.*

3. Systems with asynchronous message passing. A distributed system consists of a collection of processes and communication subsystem. Each process is a transition system in itself, with the annotation that it can interact with the communication subsystem. To avoid confusion between attributes of the distributed system as a whole and attributes of individual processes, we use the following convention. The terms 'transition' and 'configuration' are used for attributes of the entire system, and the (other equivalent) terms 'event' and 'state' are used for attributes of processes. To interact with the communication system a process has not only ordinary events (referred to as internal events) but also send events and receive events, in which a message is produced or consumed. Let $\mathcal{M}$ be a set of possible messages and denotes the collection of multisets with elements from $\mathcal{M}$ by $\mathbb{M}(\mathcal{M})$.

Definition 2.4 *The local algorithm of a process is a quintuple* $(Z, I, \vdash^i, \vdash^s, \vdash^r)$*, where* Z *is a set of states,* I *is a subset of* Z *of initial states,* $\vdash^i$ *is a relation on* $Z \times Z$*, and* $\vdash^s$ *and* $\vdash^r$ *are relations on* $Z \times \mathcal{M} \times Z$*. The binary relation* $\vdash$ *on* Z *is defined by*

$$c \vdash d \Leftrightarrow (c,d) \in \vdash^i \vee \exists m \in \mathcal{M}((c,m,d) \in \vdash^s \cup \vdash^t)$$

The relations $\vdash^i$, $\vdash^s$ correspond to state transitions related with internal, send, and receive events, respectively. In the sequel we shall denote processes by p, q, r, p_1, p_2,etc., and denote the set of processes of a system by $\mathbb{P}$. The executions of a process are the executions of the transition system $(Z, \vdash, I)$, in such an execution the executions of the processes are coordinated through the communication subsystem. To describe the coordination, we shall define a distributed system as a transition system where the configuration set, transition relation, and initial states are constructed from the corresponding components of the processes.

Definition 2.5 *A distributed algorithm for a collection* $P = p_1, ..., p_N$ *of processes is a collection of local algorithms, one for each process in P.*

The behavior of a distributed algorithm is described by a transition system as follows. A configuration consists of the state of each process and the collection of messages in transit; the transition are the

events of the processes, which do not only affect the state of the process, but can also affect (and be affected by) the collection of messages; the initial configurations are the configurations where each process is in an initial state and the message collection is empty.

Definition 2.6 *The transition system induced under asynchronous communication by a distributed algorithms for processes $p_1, ...p_k$ (where the local algorithm for process p_i is $(Z_{p_i}, I_{p_i}, \vdash^i_{p_i}, \vdash^s_{p_i}, \vdash^r_{p_i})$), is $S = (\mathcal{C}, \rightarrow, \mathcal{I})$ where*

(1) $\mathcal{C} = (c_{p_1}, ..., cp_N, M) : \forall p \in Z_p and M \in \mathbb{M}(\mathcal{M})$

(2) $\rightarrow = (\cup_{p \in \mathbb{P}} \rightarrow_p)$*, where* $\rightarrow_p$ *are the transitions corresponding to the state changes of process p;* $\rightarrow_p$ *is the set of pairs* $(c_{p_1}, ..., c_{p_i}, ..., c_{p_N}, M_1)$*,* $(c_{p_1}, ..., c'_{p_i}, ..., c_{p_N}, M_2)$ *for which one of the following three conditions holds:*

(a) $c_{p_1}, c_{p_i} \in \vdash^i_{p_i}$ *and* $M_1 = M_2$*;*

(b) for some $m \in \mathcal{M}$*,* $(c_{p_i}, m, c'_{p_i}) \in \vdash^s_{p_i}$ *and* $M_2 = M_1 \cup m$*;*

(c) for some $m \in \mathcal{M}$*,* $(c_{p_i}, m, c'_{p_i}) \in \vdash^r_{p_i}$ *and* $M_2 = M_1 \cup m$*;*

(3) $\mathcal{I} = (c_{p_1}, ..., (c_{p_N}, M)) : (\forall_p \in \mathbb{P} : c_p \in I_p) \wedge M = \emptyset$

An execution of the distributed algorithm is an execution of this induced transition system. The events of an execution are made explicit with the following notations. The pairs$(c, d) \in \vdash^i_p$ are called (possible) internal events of process p, and the triples in $\vdash^s_p \; and \; \vdash^r_p$ are called the send and receive events of the process.

(1) An internal event e given by $e = (c, d)$ of p is called applicable in configuration $\gamma = (c_{p_1}, ..., c_p, ..., c_{p_N}, M)$ if $c_p = c$. In this case, $e(\gamma)$ is defined as the configuration $c_{p_1}, ..., c_p, ..., c_{p_N}, M)$.

(2) A send event e given by $e = (c, m, d)$ of p is called applicable in configuration $\gamma = (c_{p_1}, ..., c_p, ..., c_{p_N}, M)$ if $c_p = c$. In this case, $e(\gamma)$ is defined as the configuration

It is assumed that for each message there is a unique process that can receive the message. This process is called the destination of the message.

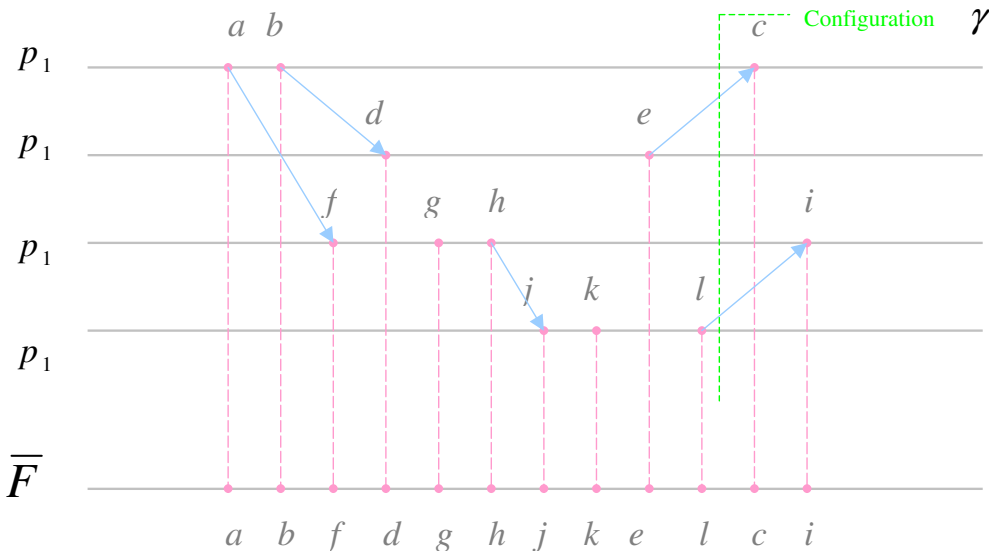

Figure 2.1: A space-time diagram

4. Causal order of events and logical clocks. The view on executions as sequences of transitions naturally induces a notion of time in executions. A transition a is then said occur earlier than transition b if a occurs in the sequence before b. For an execution $E = (\gamma_0, \gamma_1, ...)$, define the associated sequence of events $E = (e_0, e_1, ...)$, where e_i is the event by which the configuration changes from γ_i to γ_{i+1}. Observe that each execution defines a unique sequence of events in this way. An execution can be visualized in a space-time diagram, of which figure 2.1 presents and example. In such a diagram, a horizontal line is drawn for each process, and each event is drawn as a dot on the line of the process where it takes place, If a message m is sent in event s and received in event r, an arrow is drawn from s to r; the events s and r are said to be corresponding events in this case.

   Events of a distributed execution can sometime be interchanged without affecting the later configurations of the execution. Therefore a notion of time as a total order on the events of an execution is not suitable for distributed executions, and instead the notion of causal dependence is introduced next.

5. Independence and Dependence of Events. It has been remarked already that the transitions of a distributed system influence, and are influenced by, only part of the configuration. This leads to the observation that two consecutive events, influencing disjoint parts of systems with asynchronous message passing. This is expressed in the following theorem.

   Theorem 2.1 *Let γ be a configuration of a distributed system (with asynchronous message passing) and let e_p be events of different processes p and q, both applicable in γ. Then $e_p(\gamma)$, and $e_p(e_q(\gamma)) = e_q(e_p(\gamma))$.*

   Proof see sectionB.0.1.

   Indeed, the theorem explicitly states that p and q must be different, and if e_q receives the message sent in e_p, the receive event is not applicable in the starting configuration of e_p , as also required. Thus, if one of these two statements is true, the events cannot occur in the reversed order; otherwise they can occur in reversed order and yet result in the same configuration. Note that from a global point of view transitions cannot be exchanged, because (in the notation of Theorem 2.1) the transition from γ_p to γ_{pq} is different from the transition from γ to γ_q. However, from the point of view of the process these events are indistinguishable.

   The fact that a particular pair of events cannot be exchanged is expressed by saying that there is a causal relation between these two events. This relation can be extended to partial order on the set of events of an execution, called the causal order of the execution.

   Definition 2.7 *Definition Let E be an execution. The relation $\prec$, called causal order, on the events of the execution is the smallest relation that satisfies.*

   (1) If e and f are different events of the same process and e occurs before f, then $e \prec f$.

   (2) If s is a send event and r the corresponding receive event, then $s \prec \gamma$.

   (3) $\prec$ is transitive.

We write $a \preceq b$ to denote $(a \prec b \vee a = b)$. As $\preceq$ is a partial order. There may exist events a and b for which neither $a \preceq b$ nor $b \preceq a$ holds. Such events are said to be concurrent, notation $a \parallel b$. In figure 2.1 $b \parallel f$, $d \parallel i$, etc.

Causal order was first defined by Lamport [Lam78] and plays an important role in the reasoning concerning distributed algorithms. The definition of $\prec$ implies the existence of a causality chain between causally related events. By this we mean that $a \prec b$ implies the existence of a sequence $a = e_0, e_1, ..., e_k = b$, such that each pair satisfying 2.7 is a consecutive pair of events in the process where they occur, i.e. there is no other event between them. In figure 2.1, a causality chain between event a and event l is the sequence a, f, g, h, k, l.

6. Equivalence of executions computations. In this subsection it is shown that the events of an execution can be reordered in an y order consistent with the causal order, without affecting the result of the execution. This reordering of the events gives rise to a different sequence of configurations, but this execution will be execution will be regarded as equivalent to the original execution.

   Let $f = (f_0, f_1, f_2, ...)$ be a sequence of events. This sequence is the sequence of events related to an execution. This reordering of the events gives rise to different sequence of configurations. but this execution will be

7. Additional assumptions, complexity. The definitions made so far in this chapter are sufficient to set the scene for the remaining chapters; the defined model serves as a framework for the presentation and verification of algorithms, as well as for impossibility proof for solutions of distributed problems. This section discuss some of this terminology, which is also common in the literature on distributed algorithms.

8. Properties of the channels. The model can be refined by representing the contents of each channel separately in the configuration, that is, replacing the set M by a collection of sets M_{pq} for each (unidirectional) channel pq. As we have postulated that each message implicitly defines its destination. This modification does not alter the important properties of the model. Next some commonly made

assumptions about the correspondence between send and receive events are discussed.

(a) *Reliability*. A channel is said to be reliable when every message that is sent in the channel is received exactly once (provided that the destination is able to receive the message). Unless stated otherwise, it is always assumed in this paper that the channel are reliable. This assumption in fact adds a (weak) fairness condition; indeed, after a message has been sent, the receipt of this message (in the suitable state of the destination) is applicable from them on.

A channel that is not reliable may exhibit communication failures, which can be of several types, e.g. loss, garble, duplication, creation. These failures can be represented by transitions in the model of definition 2.6, but these transitions do not correspond to state changes of a process.

The loss of a message occurs when the message is sent, but never received; it can be modelled by a transition that removes the message from M. The garbling of a message occurs when the message received is different from the message sent; it can be modelled by a transition that changes one message of M. The duplication of a message occurs when the message is received more often than it is sent; it is modelled by a transition that copies a message of M. The creation of a message occurs when a message is received that has never been sent; it is modelled by a transition that inserts a message in M.

(b) *The fifo property*. A channel is said to be fifo if it respects the order of the messages sent through it . That is, if p sends two messages m_1 and m_2 to process q and the sending of m_1 occurs earlier in p than the sending of m_2, then the receipt of m_1 occurs earlier in q than the receipt of m_2.

Fifo channels can be represented in the model of Definition 2.6 by replacing the collection M by a set of queues, one for each queue, and a receive event deletes a message from the head. When fifo channels are assumed, a new type of communication failure arises, namely the reordering of messages in a channel;

it can be modelled by a transition that exchanges two messages in the queue.

A weaker assumption was proposed by Ahuja [Ahu90]; A flush channel is a channel that respects the order only of messages for which this is specified by the sender. Stronger assumptions can also be defined. Schiper *et al.* [SES89] defined causally ordered message delivery as follows. If p_1 and p_2 send messages m_1 and m_2 to process q in events e_1 and e_2 and it is the case that $e_1 \prec e_2$, then q receives m_1 before m_2. A hierarchy of delivery assumptions, consisting of asynchronism, causally ordered delivery, fifo, and synchronous communication, was discussed by Charron-Bost et al. [CBMT92]

(c) *Channel capacity*. The capacity is the number of messages that can be in transit in the channel at the same time. The channel is full in each configuration in which it actually contains a number of messages equal to its capacity. A sending event is applicable only if the channel is not full.

Definition 2.6 models channels with unbounded capacity, i.e., channels that are never full. In this paper it will always be assumed that the capacity of the channels is unbounded.

9. Real-time assumptions. An essential property of the model presented is, of course, its distributiveness: the complete independence of events in different processes, as expressed in Theorem 2.1. This property is lost when a global time frame and the ability of processes to be observe physical time (a physical clock device) are assumed. Indeed, when some real time elapses, this time elapses in all processes, and this will show up on the clock of each process.

   Real-time clocks can be incorporated by equipping each process with a real-time clock variable; the elapse of real time is modelled by a transition that puts forward the clock of each process; usually, a bound on the message transmission time(the time between sending and receiving the message) is assumed in conjunction with the availability of real-time clocks. This bound can also be included in the general model of transition systems.

10. Process knowledge. Initial process knowledge is the term used to

refer to information about the distributed system that is represented in the in initial states of the processes. If an algorithm is said to rely on such information it is assumed that the relevant information is correctly stored in the processes prior to the start of the execution of the system. Examples of such knowledge including the following information.

(1) *Topological information*. Information about the topology includes: the number of processes, the diameter of the network graph, and the topology of the graph. The network is said to have a sense of direction if a consistent edge-labelling with directions in the graph is known to the process.

(2) *Process identity*. In many algorithms it is required that the processes have unique names (identities), and that each process knows its own name initially. The processes are the supposed to contain a variable that is initialized to this name, Further assumption can be made regarding the set from which the names are chosen, such as that the names are linearly ordered or that they are integers.

(3) *Neighbor identities*. If processes are distinguished by a unique name. It is possible to assume that each process knows initially the names of its neighbors. This assumption is referred to as neighbor knowledge useful for purposes of message addressing; the name of the destination of a message is given when sending a message by direct addressing . A stronger assumption is that each process knows the entire collection of process names.

11. The complexity of distributed algorithms. The most important property of a distributed algorithm is its correctness: it must satisfy the requirements posed by the problem that the algorithm is to solve. To compare different algorithms for the same problem it is useful to measure the consumption of resources by an algorithm. The lower this assumption, the 'better' is the algorithm. The resource consumption of distributed algorithms can be measured in several ways.

(1) *Message complexity*. This is the total number of messages exchanged by the algorithm

(2) *Bit complexity*. As the set M usually contains different messages, an amount of information must be transmitted in every message to identify it in M. As the set $\mathcal{M}$ usually contains different messages, an amount of information must be transmitted in every message to identify it in $\mathcal{M}$. For an algorithm that uses a small set $\mathcal{M}$ each message can be identified using only a small number of bits, while algorithms using many different messages require more bits in each message. As 'long' messages are more expensive to transmit than 'short' messages, one may also count the total number of bits contained in messages.

Most of the algorithms in this book use messages that contain $\mathcal{O}(\log(N))$ bits (where N is the number of processes), so their bit complexity exceeds their message complexity by a logarithmic factor. In most cases only the message complexity of algorithms will be analyzed, and the bit complexity will be computed only for algorithms using very long or very short messages.

(3) *Time complexity*. As this model of distributed algorithms does not contain a notion of time it is not obvious how the time complexity of distributed algorithms can be defined. Different definitions are found in the literature. The definitions are found used in this book is based on an idealized timing of the events of a computation according to the following assumptions.

   (a) The time for processing an event is zero time units
   (b) The transmission time (i.e. the time between sending and receiving a message) is at most one time unit.

   The time complexity of an algorithm is the time consumed by a computation, under these assumptions. Note that the assumptions are only made for the purpose of defining the time complexity of the algorithm. The correctness of an asynchronous algorithm must be proved independently of these assumptions.

(4) *Space complexity*. The space complexity of an algorithm equals the amount of memory needed in a process of executed it. The space in a process is the logarithm of the number of states of that process.

   As the operation of distributed algorithms is non-deterministic, an algorithm may give rise to several computations for which

these measures may not be equal. Therefore a distinction between worst-case and average-case complexity is made. The worst-case measure is the highest complexity of any computation of the algorithm. The average case, as implied, averages over all possible computations, but in order to do so a probability distribution over all computations must be defined.

2.3 Architecture

Depending on the complexity of the tasks performed by subsystems (nodes) of a distributed system may require that this subsystem is designed in a structured way. Software for implementing this kind of distributed networks usually structured in dependent modules, each performing a specific function and relying on services offered by other modules. The modules are called layers or levels in the context of network implementation. See figure 2.2. Defining the number of layers and the interfaces is important when designing a network. For a complected network, the architecture also includes the accompanying definitions of all interfaces and protocols, it is important that products of different that products of different company are compatible.

2.4 Language support

To carry on a distributed application, no matter for a actual physically distributed environment or simulation using software on a single computer requires the distributed algorithm coded in a programming language. Here I describe three main constructs needed for distributed programming language. The language I am using in this report is Python. A language of this must provide the means to express parallelism, process interaction, and non-determinism. Parallelism is required to program nodes of distributed system to execute their part of program concurrently. Communication between the nodes must also be supported by the programming language, so that nodes can interact with each other by means of exchanging messages. None-determinism allow nodes to receive a message from different nodes.

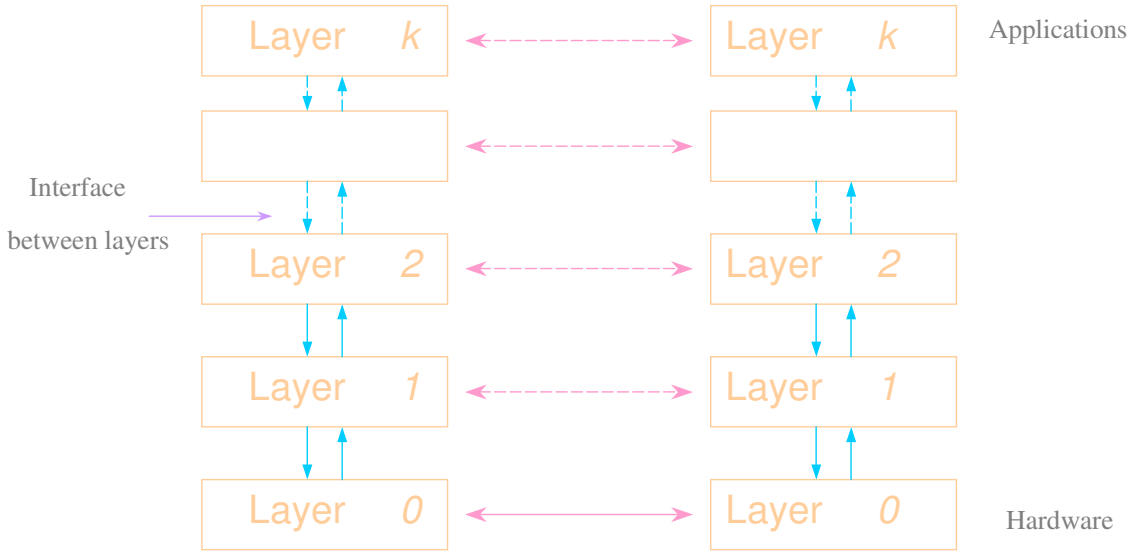

Figure 2.2: A layered network architecture

1. Parallelism. Parallelism is usually expressed by defining several processes, each being a sequential entity with its own state space. A language may either offer the possibility of statically defining a collection of processor or allow the dynamic creation and termination of processes. It is also possible to express parallelism by means of parallel statements or in a functional programming language.

2. Communication. Communication between processes is inherent to distributed algorithms: if processes do not communicate, each process operates in isolation from other processes and should be studied in isolation, not as part of a distributed system. In languages that provide message passing, 'send' and 'receive' operations are available. Communication takes places by the execution of the sent operation in one process (therefore called the *sender process*). The arguments of the send operation are the receiver's address and additional data. Forming the content of the message. The additional data becomes available to the receiver when the receive statement is executed , which implements the synchronization.

   Messages can be sent point-to-point, i.e., from one sender to one re-

ceiver, or broadcast, in which case the same message is received by all receivers. The term multicast is also used to refer to messages that are sent to a collection of (not necessary all) processes.

An alternative for message passing is the use of share memory for communication; one process writes a value to a variable, and another process reads the value.

3. Non-determinism At many points in its execution a process may be able to continue in various different ways. A receive operations is often non-deterministic because it allows the receipt of messages from different senders. Additional ways to express non-determinism are based on guarded commands. A guarded command in its most general form is a list of statements, each preceded by a boolean expression (its guard). The process may continue its execution which any of the statements for which the corresponding guard evaluates to true. A guard may contain a receive operation, in which case it evaluates to true if there is a message available to be received.

   In general, four steps are involved in performing a computational problem in parallel. The first step is to understand the nature of computations in the specific application domain. The second step involves designing a parallel algorithm or parallelizing the existing sequential algorithm. The third step is to map the parallel algorithm in a suitable parallel computer architecture, and the last step involves writing the parallel program utilizing an applicable parallel programming approach.

Chapter 3

Basic algorithms and related works

As a first approach to the graph center problem three basic algorithms will be discussed in this chapter, for general networks. These algorithms was carefully chosen for two reasons. First of all, they are asynchronous, and apply to arbitrary network. For distributed systems existing algorithms usually cover only particular types of networks and topologies. The second reason is because of their simplicity, we could compute and test them on small networks for better understand. As an additional reason to choose the leader election algorithm is because it terminate itself. In the end of this chapter some other related works will be provided.

3.1 Test connectivity

The problem that we treat in this section is the problem of discovery, by each node in N, of the identifications of all the other nodes to which it is connected by a path in G. The relevance of this problem becomes apparent when we consider of the practical situations in which portions of G may fail, possibly disconnecting the graph and thereby making unreachable form each other a pair of nodes that could previously communicate over a path of finite number of edges. The ability to discover the identifications of the nodes that still share a connected component of system in an environment that is prone to such changes my be crucial in many

cases. The algorithm that we present in this section is not really suited to the case in which G changes dynamically. The treatment of such cases requires techniques that are altogether absent from this book where we take G to be fixed and connected.

The algorithm is called *A_Test_Connectivity*, and its essence is the following. First of all, it may be started by any of the nodes in N, either spontaneously (if the nodes is in N_0) or upon receipt of the first message (otherwise). In either case, what a node n_i does to initiate its participation in the algorithm is to broadcast its identification, call it id_i, As we will see, this very simple procedure, coupled with the assumption that the edges in G are FIFO, suffices to ensure that every node in N obtains the identifications of all the other nodes in G.

The set of variables that node n_i employs to participate in this algorithm includes $parent_i^j$, $count_i^j$, $reached_i^j$, $initiated_i^j$ for $n_j \in N$

> *parent*$_i^j$, initialized to **nil**, indicates the node the node in $Neighi$ from which the first id_j has been received.
>
> *count*$_i^j$,initialized to zero, stores the number of times id_j has received.
>
> *reached*$_i^j$, initialized to **false**, is used to indicate whether id_j has been received at least once.
>
> *initiated*$_i^j$, initialized to **false**, is employed at n_i to indicated whether $n_i \in N_0$.

This algorithm is based the assumption that G's edges are FIFO. The wave that node n_j propagates forward with its identification reaches every other node n_j either when $initiated_j =$ **true** or when $initiated_j =$ **false**. By *Actions*, and because of the FIFO property of the edges, in either case id_j is only sent along the nodes on the path from n_j to n_i obtained by successively following the $parent$ pointers after id_j from all of its neighbors it has already received id_j once. Because this is valid for all $n_j \in N$, then n_j must by this time know the identifications of all nodes in G.

Algorithm *A_Test_Connectivity*:

▷ **Variables:**

$parent_i^k =$ **nil** for all $n_k \in N$;
$reached_i^k = count_i^k = 0$ for all $n_k \in N$;
$initiated_i^k =$ **false**;

▷ **Input:**

$msg_i =$ **nil**

Action:

if $n_i \in N_0$:

$initiated_i^k =$ **true**;
$reached_i^k =$ **false**;
Send id_i to all $n_j \in Neig_i$.

▷ **Input:**

$msg_i = id_k$ such that $origin_i(msg_i) = (n_i, n_j)$ for some $n_k \in N$.

Action:

if not $initiated_i$ **then**

$initiated_i :=$ **true**;
$reached_i :=$ **true**;
Send id_i to all $n_l \in Neih_i$.

$count_i^k := count_i^k + 1$;
if not $reached_i^k$ **then**

$reached_i^k :=$ **true**;
$parent_i^k := n_j$;
Send id_k to every $n_l \in Neigh_i$ such that $n_l \neq parent_i^k$.

if $count_i^k = |Neigh_i|$ **then**

if $parent_i \neq$ **nil then**

Send id_k to $parent_i^k$.

The complexity of Algorithm *A_Test_Connectivity* is $\mathcal{O}(nm)$ (to be precise, each edge carries exactly n messages in each direction, so the total number of messages is $2nm$). Because the lengths of messages depend upon n, it is in this case appropriate to compute the algorithm's bit complexity as well. If we assume that every node's identification can be expressed in $\lfloor \log n \rfloor$ bits, then the bit complexity of this algorithm is $\mathcal{O}(nm \log n)$.

3.2 All-pair shortest path

Another basic problem considered in this section is the problem of determining the shortest distances in G between all pairs of nodes. Distances between two nodes are in this section taken to be measured in numbers of edges, at the end a node be informed not only of the distance from it to all other nodes, but also of which of its neighbors lies on the corresponding shortest path, the availability of this information at all nodes provides a means of routing messages form every node to every other node along shortest paths. When G has one node for every processor of some distributed-memory system and its edges reflect the interprocessor connections in that system, this information allows shortest-path routing to be done.

An asynchronous algorithm *A_Compute_Distances* is provided here. It is widely used, despite having been displaced by more efficient algorithms of various theoretical interest. In addition to its popularity, good reasons for me to present it in detail are its simplicity.

The variables this algorithm employs includes $dist_i^j$, $first_i^j$, set_i, $level_i^j$, $state_i$, $initiated_i$ id_i, denotes n_i's identification.

> $dist_i^j$ denotes the shortest distance form n_i to $n_j \in N$, initially equal to n (unless $j = i$, in which case the initial value is zero)
>
> $first_i^j$ denotes the node in $Neigh_i$ on the corresponding shortest path to $n_j \neq n_i$, initially equal to **nil**. The set of identifications to be sent out to neighbors at each step is denoted by set_i; initially, it contains n_j's identification only.

set_i, denote he set of identifications to be sent out to neighbors at each step, initially it contains n_i's identification only.

$level_i^j$, denotes which set of node identifications n_i has received from n_j. Specifically, $level_i^j = d$ for some d such that $0 \leq d < n$ if and only if n_i has received from n_j the identifications of those nodes which are d edges away from n_j. Initially equal to -1.

$state_i$, $initiated_i$ is employed by n_j with the following meaning. Node n_i has received the identifications of all nodes that are d edges away from it for some d such that $0 \leq d < n$ if and only if $state_i = d$. Initially equal to 0.

$initiated_i$ initially set to $false$, is used to indicate whether $n_i \in N_0$

Algorithm *A_Compute_Distance*:

▷ **Variables:**

$dist_i^i = 0$;
$dist_i^k = n$ for all $n_k \in N$ such that $k \neq i$;
$first_i^k =$ **nil** for all $n_k \in N$ such that $k \neq i$;
$set_i = id_i$;
$level_i^k = -1$ for all $n_k \in Neigh_i$;
$initiated_i =$ **false** ;

▷ **Input:**

$msg_i =$ **nil**

Action:

if $n_i \in N_0$:

$initiated_i^k =$ **true**;
Send set_i to all $n_j \in Neigh_i$.

▷ **Input:**

$msg_i = set_j$ such that $origin_i(msg_i) = (n_i, n_j)$.

Action:

if not *initiated*$_i$ **then**

initiated$_i$ = **true**;

Send set_i to all $n_k \in Neigh_i$.

if *state*$_i$ **then**

level$_i^j$ = *level*$_i^j + 1$;

for all $id_k \in set_j$ **do**

if $dist_i^k > level_i^j + 1$

$dist_i^k = level_i^k + 1$

$first_i^k = n_j$

if $state_i \leq level_i^j$ for all $nj \in Neigh_i$ **then**

$state_i = state_i + 1$

$set_i = \{id_k \mid n_k \in N$ and $dist_i^k = state_i\}$

Send set_i to all $n_k \in Neigh_i$

In algorithm *A_Compute_Distances*, $N_0 = N$. If $initiated_i = true$, then the second action is only executed if $state_i < n - 1$. The point to notice is that this is in accord with the intended semantics of $state_i$, because if $state_i = n - 1$ then n_i has already received the identifications of all nodes in N, and is then essentially done with its participation in the algorithm.

Another important point to be discussed right away with respect to algorithm *A_Compute_Distances* is that the FIFO property of edges, in this case, is essential for the semantics of the $level$ variables to be maintained. In the second *Action*, the distance from n_i to n_k is updated to $level_i^j + 1$ upon receipt of id_k in a set from a neighbor n_j of n_i only because that set is taken to contain the identifications of nodes whose distance to n_j is $level_i^j$. This cannot be taken for granted, though, unless (n_i, n_j) is a FIFO edge.

The complexities of algorithm *A_Compute_Distances* can also be obtained right away. By *Actions*, what node n_i does is to send its identification to all of its neighbors, then the identifications of all of its neighbors get sent, then the identifications of all nodes that are two edges away from it, and so on. Thus n_i sends n message to each of its neighbors, and the total number of messages employed is then $2nm$, yielding a message complexity of

$\mathcal{O}(nm)$ and a bit complexity of $\mathcal{O}(nm \log n)$ if node identifications can be represented in $\lfloor \log n \rfloor$ bits.

3.3 Leader election

In this section the problem of election, also called leader finding, will be discussed. The election problem was first posed by LeLann [G.L77], who also proposed the first solution. The problem is to start from a configuration where all processes are in the same state, and arrive at configuration where exactly one process is in state leader and all other processes are in the state lost.The process in state leader at the end of the computation is called the leader and is said to be elected by the algorithm.

An election under the processes must be held if a centralized algorithm is to be executed and there is no a priori candidate to serve as the initiator of this algorithm. For example, this could be the case for an initialization procedure that must be executed initially or after a crash of the system. Because the set of active processes may not be known in advance it is not possible to assign one process once and for all to the role of leader.

Definition 3.1 *An election algorithm is an algorithm that satisfies the following properties.*

(1) Each process has the same local algorithm.

(2) The algorithm is decentralized, i.e., a computation can be initialized by an arbitrary non-empty subset of the processes.

(3) The algorithm reaches a terminal configuration in each computation, and in each reachable terminal configuration there is exactly one process in the state leader and all the other processes are in the state lost.

The last property is sometimes weakened to require only that exactly one process is in the state leader. It is then the case that the elected process is aware that it has won the election, but the losers are not (yet) aware of their loss.

3.3.1 Assumption made in this section

The election problem has been studied in this section under assumptions that we now review.

(1) The system is fully asynchronous. It has been assumed that the processes have no access to a common clock and that the message transmission times can be arbitrarily long or short.

(2) Each process is identified by a unique name, its identity, which is known to the process initially. For simplicity it has been assumed that the identity of process p is just p. The identities are drawn from a totally ordered set $\mathcal{P}$, i.e., a relation $\leq$ on identities is available. The number of bits that represent an identity is w.

Theorem 3.1 *If A is a centralized wave algorithm using m messages per wave, the algorithm $Ex(A)$ elects a leader using at most nm messages.*

Proof see section B.0.2

Lemma 3.1 *Let C be a wave with one initiator p and, for each non-initiator q. Let $father_q$ be the neighbor of q from which q received a message in its first event. Then the graph $T = (P, E_T)$, with $E_T = qr : q \neq p \wedge = father_q$ is a spanning tree directed towards p.*

Proof see section B.0.3.

The event f in the third clause of Definition 6.1 can be chosen to be a send event for all q other than the process where the decide event takes place.

Lemma 3.2 *Let C be a wave and $d_p \in C$ a decide event in process p. Then*

$$\forall q \neq p : \exists f \in C_q : (f \preceq d_p \wedge f$$

is a send event)

Proof see section B.0.4.

3.3.2 Extinction and a fast algorithm

The set of variables that process p employs to participate in this algorithm includes caw_p, rec_p, $father_p$, $lrec_p$, win_p, $state_p$.

caw_p, initialized to **None**, indicates the currently active wave.

rec_p, initialized to 0, indicates the number of **tok** message received by p

$lrec_p$, initialized to **None**, indicates father in currently active wave.

win_p, initialized to 0, indicates the number of **ldr** message received by p.

$state_p$, initialized to **sleep**, indicates the status of p. If p is the leader elected, it will be **leader**, otherwise, **lost**.

$father_p$,

Algorithm *A_Leader_elect*:

▷ **Variables:**

$caw_p =$ **nil**
$rec_p = 0$;
$father_i =$**nil**
$lrec_i = 0$
$win_i =$**nil**
$initiator_i =$ **true**

▷ **Input:**

$msg_p =$ **nil**

Action:

if $initiator_p$ **then**

$caw_p = id_p$
Send $\langle$**tok**, $id_p\rangle$ to all $n_j \in Neigh_i$

▷ **while** $lrec_p < \#Neigh_p$ **do:**

Input:

$msg_p = \langle ldr, r_id \rangle$

Action:

if $lrec_p = 0$ **then**

$lrec_p = lrec_p + 1$
$win_p := id_r$
Send $\langle$**ldr**$, id_p\rangle$ to all $n_k \in Neigh_i$

Input:

$msg_p = \langle$**tok**$, id_r\rangle$

Action:

if $r_{id} < caw_p$ **then**

$caw_i = r_{id}$
$rec_p = 0$
$father_p = q$
Send $\langle$**tok**$, id_r\rangle$ to all $n_s \in Neigh_i, n_s \neq n_q$

if $r_id = caw_p$ **then**

$rec_p = rec_p + 1$

if $rec_p = \#Neig_p$

if $caw_p = id_p$ **:** send $\langle$**ldr**$, id_p\rangle$ to all $n_s \in Neigh_i$
else: send $\langle$**tok**$, caw_p\rangle$ $father_p$

(* **if** $id_r > caw_p$ **:** ignore *)

▷ **If** $win_p = id_p$ **:** $state_p = leader$
else: $state_p = lost$

An algorithm for leader election can be obtained from an arbitrary centralized wave algorithm each initiator starts a separate wave; the messages of the wave initiated by process p must be tagged with p in order to distin-

guish them from the messages of different waves. The algorithm ensures that, no matter how many waves are started, only one wave will run to a decision, namely, the wave of the smallest initiator. All other waves will be aborted before a decision can take place.

For a wave algorithm A, the election algorithm $Ex(A)$ is as follows. Each process is active in at most one wave at a time; this wave is its currently active wave, denoted caw, with initial value undefined. Initiators of the election act as if they initiate a wave and set caw to their own identity. If a message of some wave, say the wave initiated by q, arrives at p, p processes the message as follows. If $q > caw_p$, the message is simply ignored, effectively causing q's wave to fail. If $q = caw_p$, the message is treated exactly according to the wave algorithm. If $q < caw_p$ or caw_p is undefined, p joins the execution of q's wave by resetting its variables to their initial values and setting $caw_p = q$. When the wave initiated by q executes a decision event. q will be elected.

3.4 Summary of papers

The summary of some existing algorithm are summarized as follows.

Algorithms of leader election				
Paper Name	$\|n\|$	Dyn	Asyn	Assumptions or others
[MWV00] Leader election algorithms for mobile ad hoc networks	not	$\surd$	not	Each connected component is a leader- oriented DAG originally .Maintain a unique leader for every component
[NOb] Uniform leader election protocols for radio networks[a]	not	not	not	Uniform[b], single-hop [c], single-channel[d]. when two or more stations are transmitting on a channel in the same time slot, the corresponding packages collide and are garbled beyond recognition. [e]
[NOa] Randomized leader election protocols for Ad-hoc networks	$\surd$	not	not	A fast anonymous probabilistic algorithm Elect a leader in a n-station, single-channel ad-hoc network
[NO00] Randomized leader election protocols in radio networks with no collision detection	not	not	not	Energy-efficient randomized leader election protocols for single-hop, single channel radio network. That do not have the collision detection capabilities.

[a] A radio network is a distributed system with no center arbiter, consisting of n radio transceivers, henceforth referred to as stations

[b] A leader election protocol is said to be uniform if in each time slot every station transmits with the same probability.

[c] A radio network is said to be single hop when all the station are within transmission range of each other.

[d] the stations communicate over a unique radio frequency channel known to all the stations

[e] Collision detection, in the radio network with collision detection the status of a radio channel in a time slot is, *null* if no station transmitted in the current time slot, *single* if exactly one station transmitted in the current time slot, *collision* if two or more stations transmitted the channel in the current time slot)

Routing protocols for multi-hop wireless ad hoc network	
Protocol name	abstract
[Per97] Destination-sequenced distance vector	**DSDV** is a hop-by-hop distance vector routing protocol requiring each node to periodically broadcast routing updates. It guarantees loop-freedom
[GB99] Ad-Hoc on-demand distance vector routing	**AODV** is a combination of both **DSR** and **DSDV**, It uses the on-demand mechanism of Route Discovery and Route Maintenance from **DSR**, plus the use of hop-by-hop routing, sequence numbers, and periodic beacons from **DSDV**.
[JM96]Dynamic source routing	**DSR** uses source routing rather than hop-by-hop routing, with each packet to be routed carrying in its header the complete, advantage of source routing is that intermediate nodes do not need to maintain up-to-date routing information in order to route the packets they forward, since the packets themselves already contain all the routing decisions.
[PC97] Temporally-Ordered routing algorithm	**TORA** Discover routes on demand, provide multiple routes to a destination, establish route quickly, and minimize communication overhead by localizing algorithmic reaction to topological changes when possible. (shortest-path-path routing is the secondary importance, and longer routes are often used to avoid the overhead of discovering newer routes.)

Other documents		
Name	author	feature
Netchange algorithm	Tajibnapis [Taj77]	This algorithm computes routing tables that are optimal according to the 'minimum-hop' measure. Maintains information that allows the tables to be updated with only a partial recomputation after the failure or repair of a channel. The algorithm can handle the failure and repair or addition of channels, but it is assumed that a node is notified when adjacent channel fails or recovery of nodes is not considered; instead it is assumed that the failure and recovery of node is observed by its neighbors as the failure of the connecting channel. The algorithm maintains in each node u a table of $Nb_u[v]$, giving for each destination v a neighbor of u to which packets for v terminates after a finite number of steps in all cases because the failure or repair of channels may ask for recomputation indefinitely. The requirements of the algorithm are as follows.
Locationing in distributed ad-hoc wireless sensor networks	Chris Sacarese [SR], Jan M. Rabaey	The algorithms presented herein rely on range measurements between pairs of nodes and the a priori coordinates of sparsely located anchor nodes. Clusters of nodes surrounding anchor nodes cooperatively establish confident position estimates through assumptions, checks, and iterative refinements. Once established, these positions are propagated to more distant nodes, allowing the entire network to create an accurate map of itself.
Algorithm visualization for distributed environments	Yoram Moses [MPT98]	Since in asynchronous distributed systems there is no way of knowing the 'the real' execution, a system VADE is designed to be consistent with the execution of the distributed algorithm, so that the algorithm runs on the server's machines while the visualization is executed on a web page on the client's machine
Self-stabilizing algorithms for finding centers and medians of trees	[KPBG94]	Tree graph, no initialization is assumed, the underlying network topology is permitted to change under the condition that the topology remain connected and acyclic. Self-stabilizing Due to a transient failure, a process may temporarily have incorrect knowledge about the identification of neighbors, but eventually identify its neighbors correctly and the system converge to desired state.

Chapter 4

A new algorithm for the graph center problem

Locating centers of graphs has a wide variety of important application because placing a common resource at a center of a graph minimizes costs of sharing the resource with other locations.

In this chapter I am going to introduce my own algorithm for the graph center problem. Details about center problem and definition see page 3. This is a new algorithm based on the three algorithms discussed in the previous chapter. It is an asynchronous algorithm and apply to arbitrary networks. The main feature of this algorithm is its layered structure. The main idea of this algorithm is to find the center node in a distributed environment.

4.1 Assumptions

A distributed system consists of n nodes, denoted by N_0, N_1, N_2, ..., N_3. Nodes are identical, each have a unique name. Each node have three different processors or CPUs, whose program is composed of *atomic* steps. These three processors corresponding to three different layer of the algorithm, see figure 4.1 for node hardware structure. The three layers are namely test connectivity, compute eccentricity and compute center. Each processor has its own buffer for message passing. The number of nodes is needed. Each node keeps information of its neighbours and they only

can communication with their neigbours directly. By saying neighbours, we mean nodes in distance within a limited length of the node. The cost is the number of hops (or edges) needed to the destination. The links between nodes is reliable. The links and buffers are in order of FIFO. There is indefinite delay of messages passing.

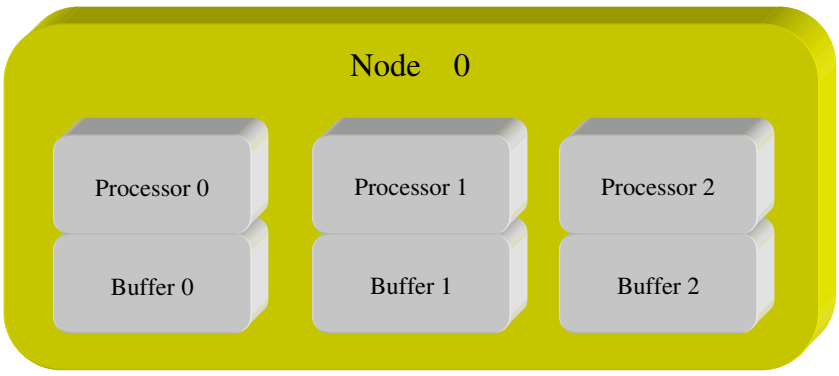

Figure 4.1: Node hardware architecture and naming method

4.2 Layered structure

This a algorithm is designed to find the node which has the maximum shortest distance to all the other nodes (called a MinMaxMin problem). Due to the complexity of the tasks performed by the distributed system, design a single algorithm can be very complicated. A algorithm designed in a structured way is required. The process flow graph for this algorithm is showed in figure 4.2.

Since this algorithm is designed in a distributed way, each node performs the same algorithm, and the tasks rely on the interaction or message passing of each nodes. But messages passing only happens among the same type of processors. In the sense of the algorithm, message passing only happens within the same layer. Figure 4.3 shows the message passing between two nodes. Here I called the second layer routing layer because it comes from all-pair-shortest path problem, and routing is a second benefit after solving center problem.

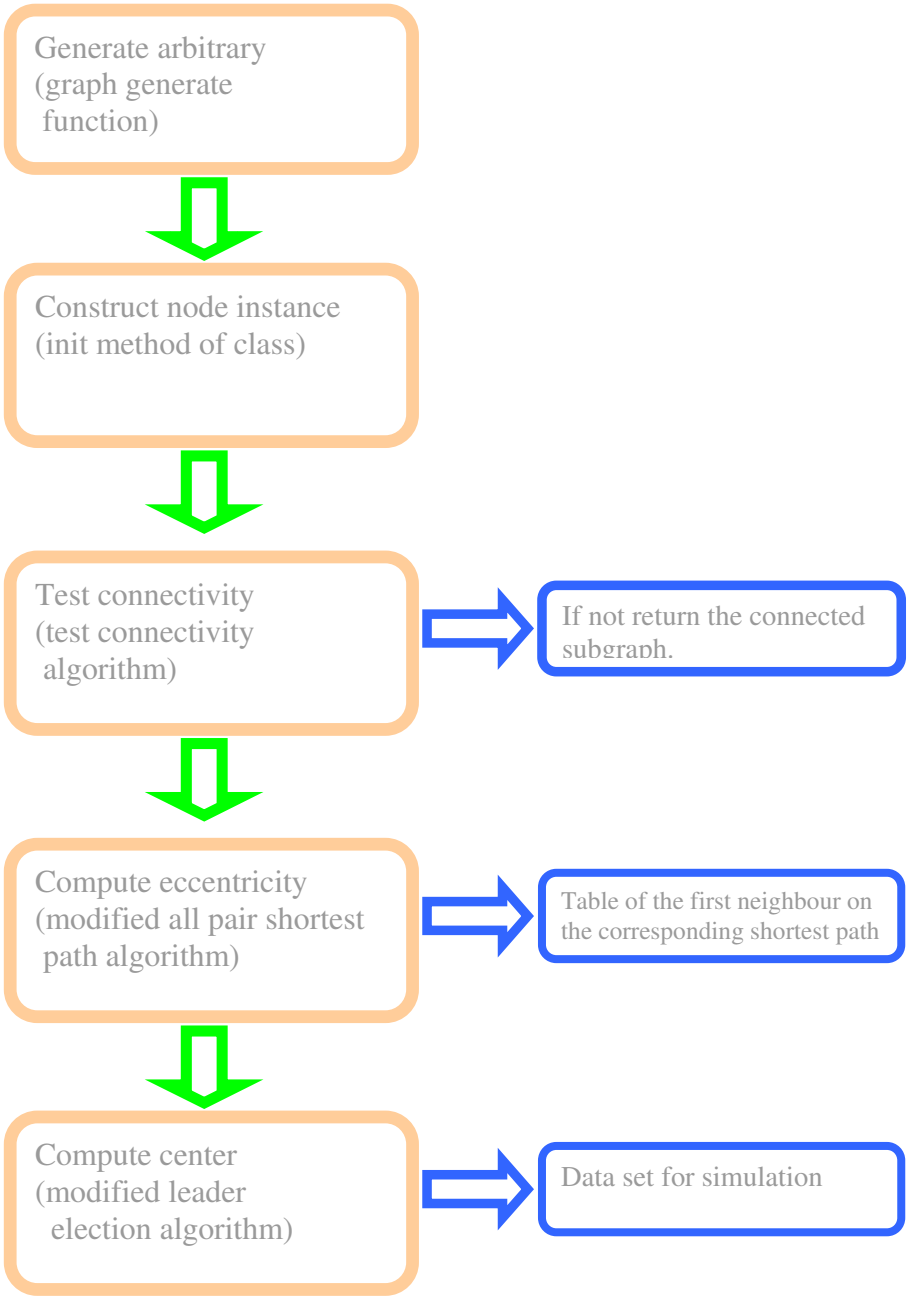

Figure 4.2: Process flow graph

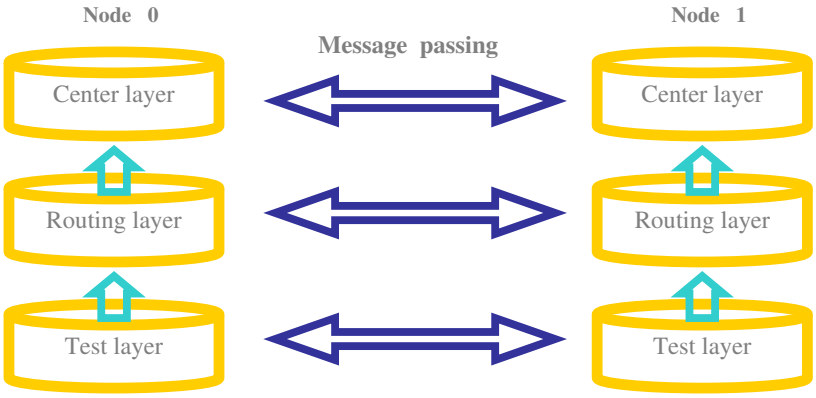

Figure 4.3: Message passing between layers between nodes

4.3 Message structure

There exists four type of messages in this algorithm. And they are structured in the same way. See figure 4.4

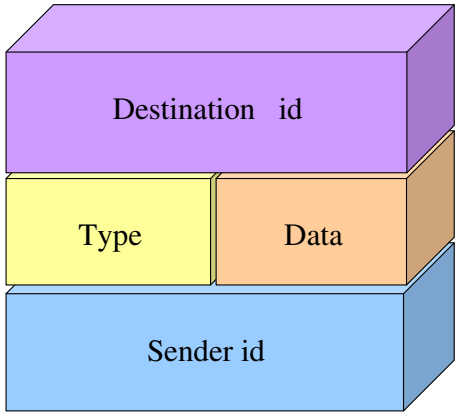

Figure 4.4: Use tuple message structure

As their names, the *Destination id* part is the address the message going to. The *type* part indicate the type of the message. This information for real computing is stored in the *data* part. After the message finds the

destination, it would be checked; this will trigger the destination node to execute different actions.

4.4 How does it work?

Why the algorithm is designed like this? How does this algorithm work? In this section first I will introduce the naming method and then the function of the different layers.

1. Naming method. As we known distributed system communicate rely on message passing. The name of node in this kind of system is a unique identification, so it can also act as the address. In this algorithm I have both nodes and processors. All of them could have their own name, but is this necessary? Actually not. The reason is as following.

   On one hand although the message is processed by processors, but there are no message passing within nodes, that is only message passing across nodes, so the first address should be the name of nodes. On the other hand the processors in the same node is not logically distributed because they have the same network topology.

   Now we have to decided after the message come to the nodes, how can it find the right processor? Remember, each processor have it own buffer, all we need to do is to send the message to the buffer corresponding to the processor.

   For ease of computing, I named the buffers and processor (they are one entity) in sequential order. And name the node as same as the first processor, *test processor*. See figure 4.1. Now we can back to see the message structure again. Of course there are always better way of doing things.

2. Layer function. When the message gets to the processor, the next step is to be processed. And before this the message need to be created, that is, the data is available. This is why the algorithm is layered and the order is acyclically designed, because they depend on the lower layer to provide information for them to process, otherwise

they needed to continue the computation, otherwise, they will stay doing nothing.

As is shown is figure 4.5. The test layer is first started, it test whether the network is connected, if not it will return the connected part of the network, since it may several partitions, I let it only return the partition, which contains node0. When this is done. The second layer is started, and compute the eccentricity for each node. This information will be passed on to the next layer to compute the center.

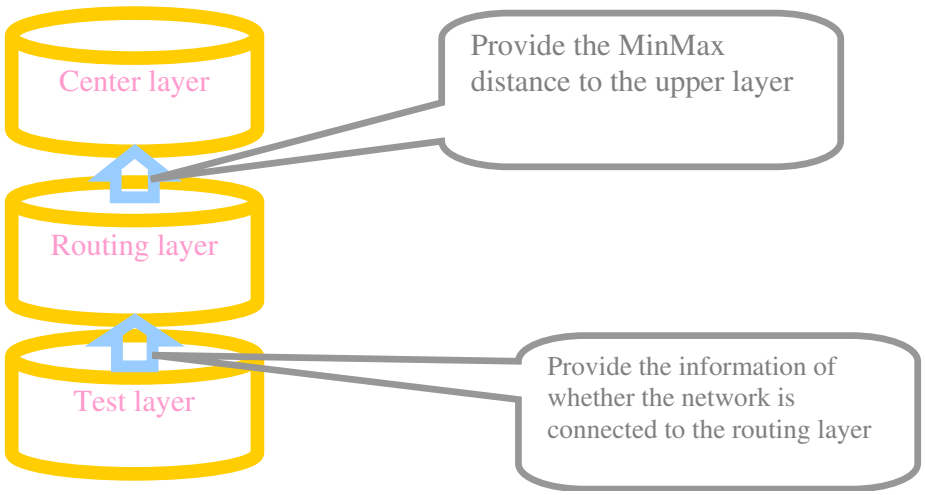

Figure 4.5: Message passing between layers inside the node

4.5 Modification of algorithms

Now as a example of message structure, I am going to explain how I modified those algorithms described in chapter 3.

1. Leader election. The main idea of that algorithm consists in assigning unique ids from an ordered set to each of the nodes in the network and then elect leader the node with the minimum id. But out aim is to find the center of a graph, we still will give each node a

unique id, but this time the problem will be compare the maxmin distance. As I said in the message structure section, because the actual computation information is stored in the *data* part.

In the leader election algorithm there exist two different type of messages, **tok** and **ldr**. Both of carry node identification, because find the node with the node with smallest id. If I want to find the center, I need to find the node with smallest eccentricity. So first I give the node a new variable d_i, initially set to 0, indicates, the eccentricity of the node. Then I change those part, where the computation make decision(at the time **ldr** message is sent). The details of the changes is as following:

(1) **Action:**

if $initiator_p$ **then**

$caw_p = id_p$

Send $\langle$**tok**$, id_p\rangle$ to all $n_j \in Neigh_i$

this is modified to

Action:

if $initiator_p$ **then**

$caw_p = d_p$

Send $\langle$**tok**$, d_p\rangle$ to all $n_j \in Neigh_i$

Reason: This will let the nodes broadcast the information which contains the value of eccentricity.

(2) **if** $caw_p = id_p$ **:** send $\langle$**ldr**$, id_p\rangle$ to all $n_s \in Neigh_i$
is changed to

if $caw_p = d_p$ **:** send $\langle$**ldr**$, id_p\rangle$ to all $n_s \in Neigh_i$

Reason: The computation here is need to make the decision, since it is going to send the **ldr** message, so change it to let the node make the decision according to the eccentricity.

(3) **If**$win_p = id_p$ **:** $state_p = leader$
is changed to
If$win_p = d_p$ **:** $state_p = leader$

Reason: The variable **win** contains the winner's information, this is a big decision. Change it to return the smallest eccentricity. Compare this value with self's eccentricity, if it is the same, then win.

2. Shortest pathFor this algorithm I changed it to return the maximum shortest distance instead of only shortest distance matrix. This is a computation done locally, I will not discuss it in details here.

4.6 Technique

The language I am used for this project is Python; python is a high-level object-oriented language, and it is programmed using Linux. List of main algorithm could be find in the appendix. Python is easy to use and very flexible. You can find most useful information from its official website `www.python.org`. Next I am going to introduce some main issues of how to use python to change the algorithm into program. For more examples, see Appendix A.

1. Using class to construct node. Python is a objected oriented language, this make it possible to make new object. *Class* is the model doing this job.

2. Using *thread* for parallel computing . Python provide a parallel computing module called *thread*. It allow several processes to run together to solve the problem. This make it possible to simulate performance of a distributed system on one computer.

3. Use *tuple* as message structure. I mentioned the message structure in former section, See figure 4.4. This could be done in Python using *tuples*. Just write it as $(destination.id, ('type', data), sender.id)$.

4. Use *list* as stack. List is another example, the list methods make it very easy to use as a list as a stack, where the last element added is the first element retrieved (FIFO), To add an item to the top of the stack, use append(). To retrieve an item from the top of the stack, use pop(0). As is showed in figure 4.6

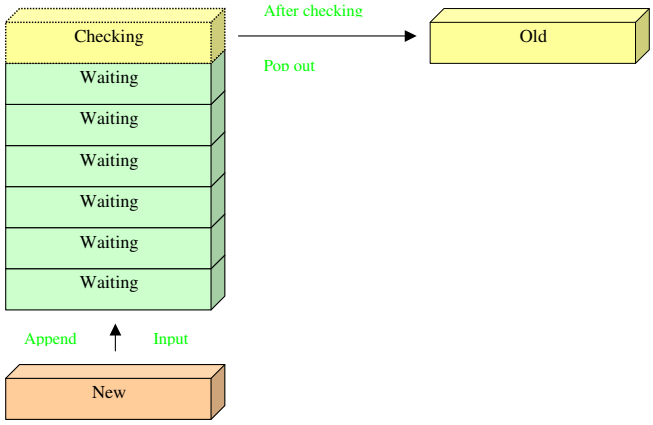

Figure 4.6: Use list message structure

4.7 Generating large graphs

As you maybe noticed the network used in this program is generated by the automatically by the program. This is not part of the algorithm, but it is there to make the program a complete implementation. See code lines 191-221 on page 60.

The idea is first I generate the network as a empty set. When construct instance of Node class. Just choose neighbours (a subset) from the existing set, so that a new instance is constructed with neighbours information. This subset may be empty. And this is want we want. Figure 4.7 is generated by this method.

4.8 How to use the code?

This program is a complete simulation of the graph center algorithm I introduced in this chapter. The execution of the program gives four dot files and the result at the same time. In this section I just simply introduce how to use it. The output of the program can be found in the section follows.

1. The latest version of Python is 2.3. You can download it from its

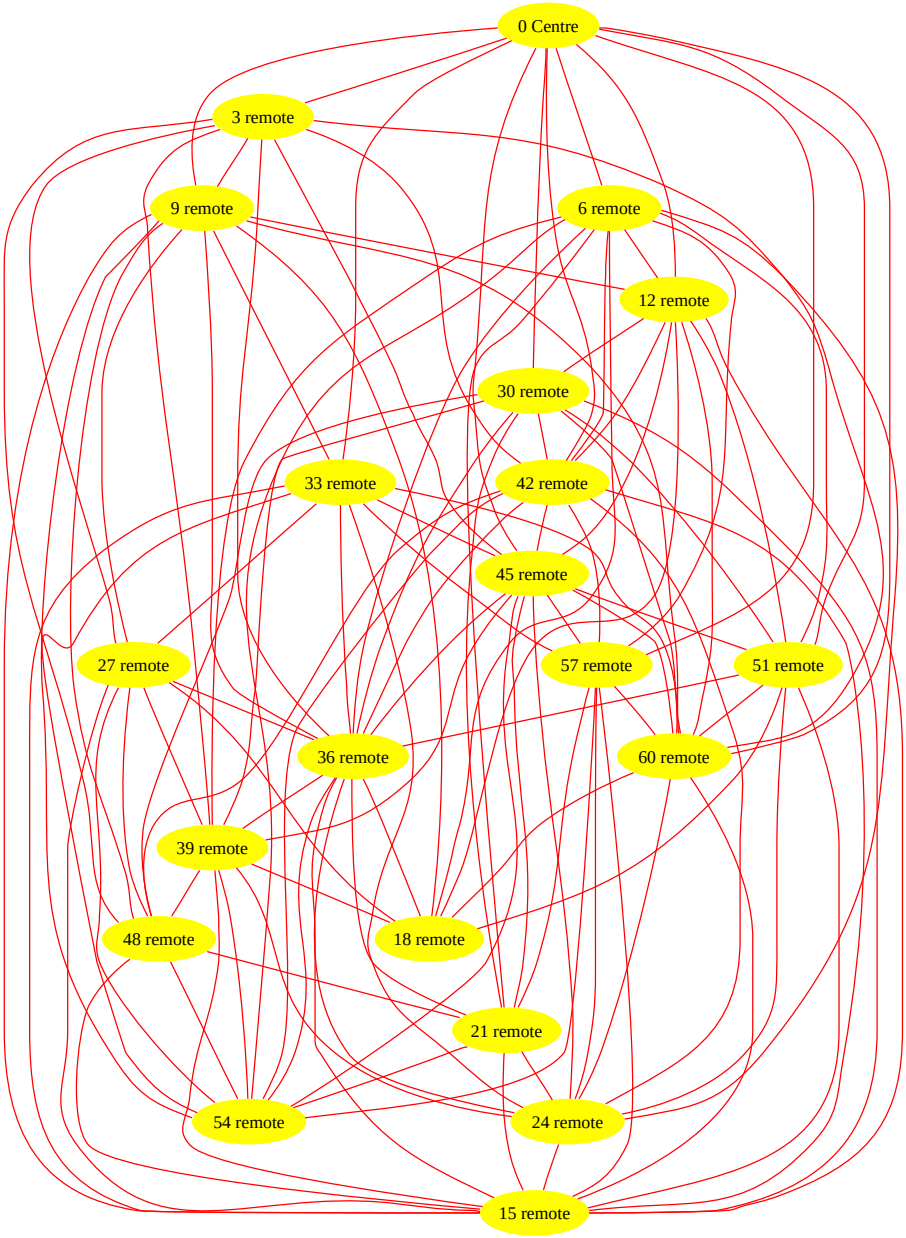

Figure 4.7: Example network with a center

official website www.python.org, and install it according to the instruction.

(a) If using platform of Windows, open the file use IDLE(Python GUI) then under **run** choose **run script**, the result will pop up from an other window, this program have a dot [1]program part with it, it will not work under windows systems, so the last line of this program need to be switch off before run.

(b) If using Linux, just type the file name from the console, the program will run and give the result of the computation, with a dot file generated in the end, follow the command line, you can open the dot file, which give you the graph the program currently working on. The file is automatically generated, and it is random. So it will be rewritten by running the program again. A program with curses output for the all-pair shortest path and a program with data set generating could be found in the next section.

(c) Change the parameter. If you would like to change the parameter to see node you can change the parameter k, but I should mention that in the first two algorithm there are no automatically termination available at this moment, so if you do a very large number, you may see the network is not connected, but this does not mean it is really not connected but not enough iteration has been set. So you need to set the parameter m to a larger number to see the right answer at the same time. The same apply to the second part of the algorithm.

[1]GraphViz provides a collection of tools for manipulating graph structures and generating graph layouts. Example applications:

- Software documentation: Pretty diagrams automatically generated by doxygen and dot. (From GraphViz source documentation.)
- WWW Graph Server: For a WWW application of GraphViz, please see http://www.graphviz.org/

4.9 Sample results

The graphs generated by the dot program part of the code, see figure 4.8

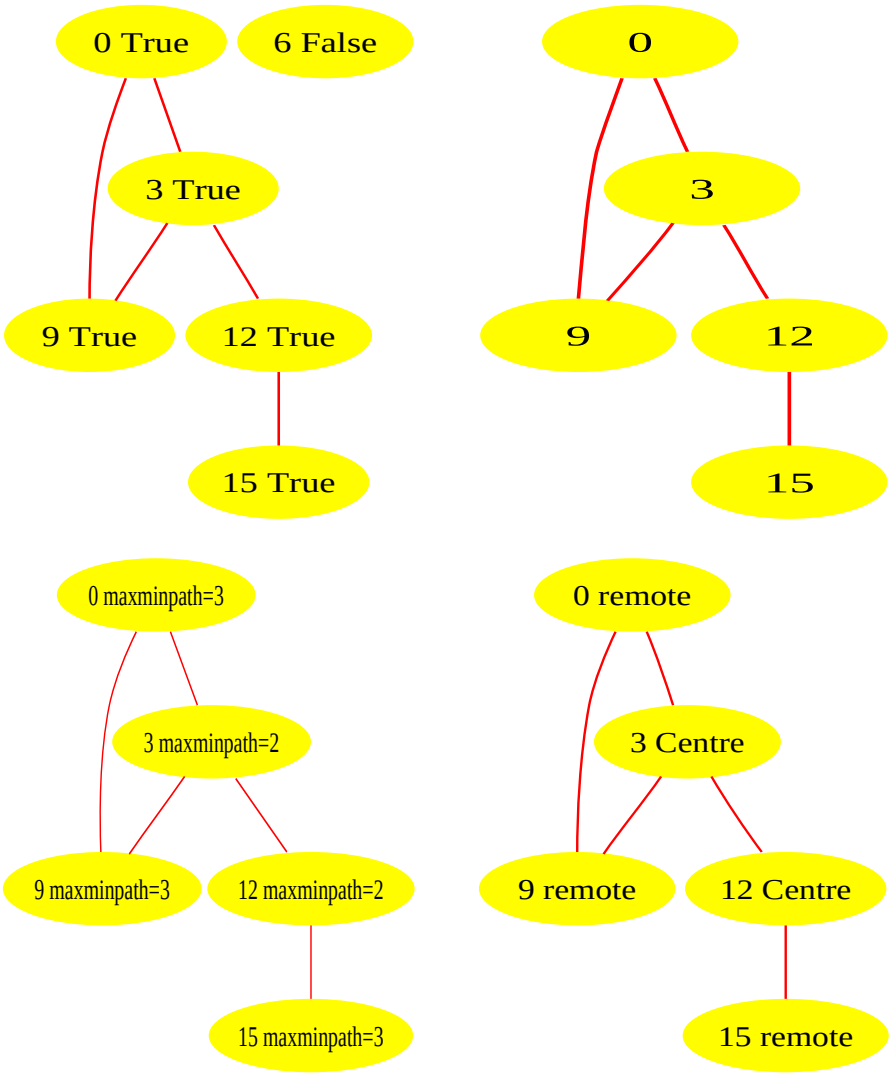

Figure 4.8: Process of the program: top left: Result of connectivity; top right: Return the connected partition; bottom left: Compute the eccentricity; bottom right: Compute the center

The result corresponding to these graphs is generated in the same time as following:

```
n0=Node([])
n1=Node([n0,])
n0.add_link(n1)
n2=Node([])
n3=Node([n0,n1,])
n0.add_link(n3)
n1.add_link(n3)
n4=Node([n1,])
n1.add_link(n4)
n5=Node([n4,])
n4.add_link(n5)
nodes=[n0,n1,n2,n3,n4,n5]

node0.reached  {0: T, 3: T, 6: F, 9: T, 12: T, 15: T}
node3.reached  {0: T, 3: T, 6: F, 9: T, 12: T, 15: T}
node6.reached  {0: F, 3: F, 6: T, 9: F, 12: F, 15: F}
node9.reached  {0: T, 3: T, 6: F, 9: T, 12: T, 15: T}
node12.reached {0: T, 3: T, 6: F, 9: T, 12: T, 15: T}
node15.reached {0: T, 3: T, 6: F, 9: T, 12: T, 15: T}

nodes [0, 3, 6, 9, 12, 15]
nodes [0, 3, 9,12, 15]

The all-pair shortest distance matrix
0  {0: 0, 9: 1, 3: 1, 12: 2, 15: 3}
3  {0: 1, 9: 1, 3: 0, 12: 1, 15: 2}
9  {0: 1, 9: 0, 3: 1, 12: 2, 15: 3}
12 {0: 2, 9: 2, 3: 1, 12: 0, 15: 1}
15 {0: 3, 9: 3, 3: 2, 12: 1, 15: 0}

Table of first neighbor corresponding to shortest path
  ____________________________________________________
0 | {0: None, 9: 9,    3: 3,    12: 3,    15: 3   }
  |
```

```
1 | {0: 0,    9: None, 3: 3,    12: 3,    15: 3   }
  |
2 | {0: 0,    9: 9,    3: None, 12: 12,   15: 12  }
  |
3 | {0: 3,    9: 3,    3: 3,    12: None, 15: 15  }
  |
4 | {0: 12,   9: 12,   3: 12,   12: 12,   15: None}

  0 is remote  3 is Centre
  9 is remote  12 is Centre
  15 is remote

  run "dot -Tps bf1.dot | gv -" to see connectivity
  run "dot -Tps bf2.dot | gv -" to see partition contains n0
  run "dot -Tps bf3.dot | gv -" to see maxminpath
  run "dot -Tps bf4.dot | gv -" to see center
```

A sample data set for graphic interface.

4.10 Estimation and Future works

This algorithm solved the graph center problem for distributed environment. When building such systems, including asynchrony, we have to cope with two other main issues; termination and failure occurrence.

Failure occurrences cannot be predicted. The net effect of asynchrony and failure occurrences actually create an uncertainty on the state of the application (as perceived by a process) that can make very difficult or even impossible to determine a system view that can be validly shared by all non-faulty processes. The mastering of such an uncertainty is one of the main problems that designers of asynchronous systems have to solve.

Detecting termination of a distributed algorithm : In the of the algorithms considered in chapter 3, there are situations where the computation can naturally be viewed as terminated. For example, the leader election comes to an end when the leader is elected . Here we describe an method for detecting termination.

We consider situations where during execution of the algorithm, each processor is able to monitor its own computations and decide whether a certain 'local termination condition' holds. if the local termination condition holds at some processor, then no messages can be transmitted by that processor. Further more, once true, the local termination condition remains true until a message from some other processor is received.

Termination has occurred at some time t if:

(a) The local termination condition holds at all processors at time t.

(b) No message is in the transit along any communication link at time t.

Termination occurs at time $\bar{t}$, if $\bar{t}$ is the smallest time t for which the above conditions (a) and (b) hold. Our objective is to detect the termination within finite time after it occurs. Notice that if termination has occurred at some time t, then the same is true for every subsequent time $t' > t$, since no messages will be transmitted after time t and local termination condition will remain true at all processors.

```
0 |

[(0.00079798698425292969, 'change', ['caw', 0]),
((0.00085890293121337891, 'send', 1), ('tok', 0, 0)),
(0.013808012008666992, 'check'),
((0.01385200023651123, 'savemsg'), ('tok', 1, 1)),
(0.01391899585723877, 'ignore', ('tok', 1, 1)),
(0.21377801895141602, 'check'),
((0.21382296085357666, 'savemsg'), ('tok', 0, 1)),
(0.2138969898223877, 'change', ['rec', 1]),
((0.21396398544311523, 'send', 1), ('ldr', 0, 0)),
(0.23376691341400146, 'check'),
((0.23381400108337402, 'savemsg'), ('ldr', 0, 1)),
((0.23388791084289551, 'send', 1), ('ldr', 0, 0)),
(0.23394596576690674, 'change', ['lrec', 1]),
(0.23398900032043457, 'change', ['win', 0]),
(0.23405098915100098, 'change', ['state', 'Leader'])]

1 |

[(0.0013449192047119141, 'change', ['caw', 0]),
((0.0014050068664455078, 'send', 0), ('tok', 1, 1)),
((0.0014599561691284180, 'send', 2), ('tok', 1, 1)),
((0.0015159845352172852, 'send', 3), ('tok', 1, 1)),
((0.0015699863433837891, 'send', 4), ('tok', 1, 1)),
(0.014109015464782715, 'check'),
((0.014149904251098633, 'savemsg'), ('tok', 0, 0)),
(0.014213919639587402, 'change', ['caw', 0]),
(0.014256954193115234, 'change', ['rec', 4]),
(0.014297008514404297, 'change', ['father', 0]),
((0.014360904693603516, 'send', 2), ('tok', 0, 1)),
((0.014420986175537109, 'send', 3), ('tok', 0, 1)),
((0.014482975006103516, 'send', 4), ('tok', 0, 1)),
(0.014547944068908691, 'change', ['rec', 4]),
(0.033867001533508301, 'check'),
((0.033913016319274902, 'savemsg'), ('tok', 2, 2)),
(0.033980011940002441, 'ignore', ('tok', 2, 2)),
(0.053878903388977051, 'check'),
((0.053924918174743652, 'savemsg'), ('tok', 4, 4)),
(0.053990960121154785, 'ignore', ('tok', 4, 4)),
(0.073873996734619141, 'check'),
((0.073920011520385742, 'savemsg'), ('tok', 3, 3)),
(0.073985934257507324, 'ignore', ('tok', 3, 3)),
(0.11387395858764648, 'check'),
((0.11391901969909668, 'savemsg'), ('tok', 0, 3)),
(0.1139909029006958, 'change', ['rec', 4]),
(0.13386595249176025, 'check'),
((0.13391292095184326, 'savemsg'), ('tok', 0, 2)),
(0.13398492336273193, 'change', ['rec', 4]),
(0.1938709020614624, 'check'),
((0.19391500949859619, 'savemsg'), ('tok', 0, 4)),
(0.19398891925811768, 'change', ['rec', 4]),
((0.19405090808868408, 'send', 0), ('tok', 0, 1)),
(0.21417498588562012, 'check'),
((0.21422100067138672, 'savemsg'), ('ldr', 0, 0)),
((0.21429693698883057, 'send', 0), ('ldr', 0, 1)),
(0.21435296535491943, 'change', ['lrec', 4]),
(0.21439599990844727, 'change', ['win', 0]),
((0.21444392204284668, 'send', 2), ('ldr', 0, 1)),
(0.21449697017669678, 'change', ['lrec', 4]),
(0.21454095840454102, 'change', ['win', 0]),
((0.21458995342254639, 'send', 3), ('ldr', 0, 1)),
(0.21464300155639648, 'change', ['lrec', 4]),
(0.21468591690063477, 'change', ['win', 0]),
((0.21473395824432373, 'send', 4), ('ldr', 0, 1)),
(0.21478700637817383, 'change', ['lrec', 4]),
(0.21483099460601807, 'change', ['win', 0])]

2 |

[(0.0020499229431152344, 'change', ['caw', 0]),
((0.0021100044250488281, 'send', 1), ('tok', 2, 2)),
((0.0021649599075317383, 'send', 7), ('tok', 2, 2)),
(0.014743924140930176, 'check'),
((0.014789938926696777, 'savemsg'), ('tok', 1, 1)),
(0.014853954315185547, 'change', ['caw', 0]),
(0.014894008636474609, 'change', ['rec', 2]),
(0.014932990074157715, 'change', ['father', 1]),
((0.014997005136962891, 'send', 7), ('tok', 1, 2))
```

Appendix A

Program listings

A.1 Test connectivity

```
#!/usr/local/bin/python

from threading import *
from time import sleep
from sys import stdin,stderr,exit
from copy import copy
from random import choice

class Node:

  def __init__(s,neig):
    s.initiated=False
    s.t=Thread(target=s.a_test_connectivity)
    s.id='N'+str(int(s.t.getid()[7:])-1)
    s.neig=neig
    s.msgq=[]

  def getid(s):
    return s.id

  def send(node,msg):
    node.msgq.append(msg)

  def __repr__(s):
    return s.id
```

```
  def add_link(s,node):
      s.neig.append(node)

  def a_test_connectivity(s):
   m=100
   while m:
     m-=1
     sleep(0.01)
     if not s.msgq:
       s.intiated=True
       s.reached[s.id]=True
       for node in s.neig:
         Node.send(node,(s.id,s.id))
     else:
       if s.msgq:
          nj,nk=s.msgq.pop(0)
          if not s.initiated :
            s.initiated=True
            s.reached[s.id]=True
            for node in s.neig:
              Node.send(node,(s.id,s.id))
          s.count[nk]+=1
          if not s.reached[nk]:
             s.reached[nk]=True
             s.parent[nk]=nj
             for node in s.neig:
               if not node.id==nk:
                 Node.send(node,(s.id,nk))
          if s.count[nk]==len(s.neig):
            if s.parent[nk]:
                 if node.id==s.parent[nk]:
                   Node.send(node,(s.id,nk))

  def start(s):
    s.parent=dict([(x.id,None) for x in nodes])
    s.count=dict([(x.id,0) for x in nodes])
    s.reached=dict([(x.id,False) for x in nodes])
    s.t.start()

nodes=[n0,n1,n2,n3,n4,n5,n6]
No=[n0]
for node in nodes: node.start()

while 1:
```

```
  alive=0
  for node in nodes: alive+=node.t.isAlive()
  print >>stderr,'%d threads alive'%alive
  if alive==0: break
  sleep(1)

print 'n0.reached',n0.reached
```

A.2 Shortest path

```
#!/usr/local/bin/python

from threading import *
from time      import sleep
from sys       import stdin,stderr,exit
from copy      import copy
from set       import Set

class Node:

  def __init__(s,neighbours=[]):

    s.t=Thread(target=s.a_compute_distances)
    s.name='N'+str(int(s.t.getName()[7:])-1)
    s.set=Set([s.name])
    s.state=0
    s.initiated=False
    s.neighbours=copy(neighbours)
    s.msgq=[]

  def getName(s):
    return s.name

  def __repr__(s):
    return s.name

  def send(node,msg): # send messages
    node.msgq.append(msg)
```

```
def add_link(s,node): # add link to node                                    30
    s.neighbours.append(node)

def a_compute_distances(s):
 m=100000
 n=len(nodes)
 while m:
   m-=1
   sleep(0.001)
   if not s.msgq:
     if s in No:                                                            40
       s.initiated=True
       for node in s.neighbours:
         Node.send(node,(s.name,(s.set)))
   else:
     nj,setj=s.msgq.pop(0)
     if not s.initiated:
       s.initiated=True
       for node in s.neighbours:
         Node.send(node,(s.name,(s.set)))
     if s.state<n-1:                                                        50
       if 1:
         s.level[nj]+=1
         for nk in setj:
           if s.dist[nk]>s.level[nj]+1:
             s.dist[nk]=s.level[nj]+1
             s.first[nk]=nj
       x=True
       for nei in s.neighbours:
         x=x and s.state<=s.level[nei.name]
       if x:                                                                60
         s.state+=1
         s.set=Set([])
         for nk in nodes:
           if s.dist[nk.name]==s.state:
             s.set.insert(nk.name)
         s.bfr=copy(s.set)
         for nb in s.neighbours:Node.send(nb,(s.name,(s.set)))

def start(s):
  n=len(nodes)                                                              70
  s.dist=dict([(x.name,n) for x in nodes])
  s.dist[s.name]=0
```

```
    s.first=dict([(x.name,None) for x in nodes])
    del s.first[s.name]
    s.level=dict([(x.name,-1) for x in s.neighbours])
    s.t.start()


n0=Node([])
n1=Node([n0])
n2=Node([n1])
n3=Node([n1])
n4=Node([n1,n3])
n5=Node([n4])
n6=Node([n5])
n7=Node([n2])
n0.add_link(n1)
n1.add_link(n2)
n1.add_link(n3)
n1.add_link(n4)
n2.add_link(n7)
n3.add_link(n4)
n4.add_link(n5)
n5.add_link(n6)
nodes=[n0,n1,n2,n3,n4,n5,n6,n7]
No=[n0,n1,n2,n3,n4,n5]
for node in nodes: node.start()

while 1:
  alive=0
  for node in nodes: alive+=node.t.isAlive()
  print >>stderr,'%d threads alive'%alive
  if alive==0: break
  sleep(1)

print '    ',
for node in nodes: print  '%s '%node.name,
print
print '  ',33*'-'
for node in nodes:
  print node.name,'|',
  for k,v in node.dist.items(): print '%3d'%int(v),
  print
```

A.3 Leader election

```
#!/usr/local/bin/python

from threading import *
from time      import sleep
from sys       import stdin,stderr,exit
from copy      import copy
from math      import log, floor
from random    import randrange, choice
from time      import time

class Node:

  def __init__(s,neigh,d,initiator=True):

    s.caw=None
    s.rec=0
    s.father=None
    s.lrec=0
    s.win=None
    s.d=d
    s.initiator=initiator
    s.state=['state','sleep']
    s.neigh=neigh
    s.th=Thread(target=s.extinct_echo_center)
    s.id=int(s.th.getName()[7:])-1
    s.num=0
    s.rmsg=None
    s.buffer=[]

  def send(s,d,msg):
    d.buffer.append(msg)

  def check(s):
    s.rmsg=s.buffer.pop(0)
    return s.rmsg

  def start(s):
    s.th.start()

  def getName(s):
    return s.id
```

```
  def __repr__(s):
    return str(s.id)

  def add_link(s,z):
    s.neigh.append(z)

  def extinct_echo_center(s):
    addr=dict([(node.id,node) for node in nodes])
    s.num=len(s.neigh)
    if s.initiator:
      s.caw=s.d
      for q in s.neigh:
        s.send(q,('tok', s.d, s.id))
    while s.lrec<s.num:
      sleep(0.01+choice((0.001, 0.002, 0.003)))
      if s.buffer:
        name,r,qid=s.check()
        if name=='ldr':
          if s.lrec==0:
            for q in s.neigh:
              s.send(q,('ldr', r, s.id))
              s.lrec+=1
              s.win=r
        else:
          if r<s.caw:
            s.caw=r
            s.rec=0
            s.father=qid
            for p in s.neigh:
              if not p.id==qid:
                s.send(p,('tok', r, s.id))
          if r==s.caw:
            s.rec+=1
            if s.rec==s.num:
              if s.caw==s.id:
                for p in s.neigh:
                  s.send(p,('ldr',s.d,s.id))
              else: s.send(addr[s.father],('tok',s.caw,s.id))
    if s.win==s.d:
      s.state='Centre'
    else: s.state='remote'

n0=Node([],4)
n1=Node([n0],3)
```

```
n2=Node([n1],4)
n3=Node([n1],3)
n4=Node([n1,n3],3)
n5=Node([n4],4)                                                    90
n6=Node([n5],5)
n7=Node([n2],5)

n0.add_link(n1)
n1.add_link(n2)
n1.add_link(n3)
n1.add_link(n4)
n2.add_link(n7)
n3.add_link(n4)
n4.add_link(n5)                                                    100
n5.add_link(n6)
nodes=[n0,n1,n2,n3,n4,n5,n6,n7]

for node in nodes: node.start()
t0=time()
while 1:
  alive=0
  for node in nodes: alive+=node.th.isAlive()
  print >>stderr,'%d threads alive'%alive
  if alive==0: break                                               110
  sleep(1)

for node in nodes:print node.id,'is',str(node.state)
```

A.4 The full code for my center algorithm

```
#!/usr/local/bin/python
# three algorithm combined togeter

from threading import *
from time      import *
from sys       import stdin,stderr,exit
from copy      import copy
from set       import Set
from random    import randrange, choice
```

```
from math      import log, floor

t0=time()

class Node:

  def __init__(s,neigh,initiator=True):

    s.th=Thread(target=s.extinct_echo_center)
    s.caw=None
    s.rec=0
    s.father=None
    s.lrec=0
    s.win=None
    s.initiator=initiator
    s.status='sleep'
    s.id=int(s.th.getName()[7:])-1
    s.num=0
    s.buffer1=[]
    s.rmsg1=None
    s.t=Thread(target=s.a_compute_distances)
    s.set=Set([s.id])
    s.state=0
    s.initiated=False
    s.buffer=[]
    s.rmsg=None
    s.neigh=neigh
    s.d=None
    s.initiated1=False
    s.thh=Thread(target=s.a_test_connectivity)
    s.buffer2=[]


  def start2(s):
    s.parent= dict([(x.id,None ) for x in nodes])
    s.count=  dict([(x.id,0    ) for x in nodes])
    s.reached=dict([(x.id,False) for x in nodes])
    s.thh.start()

  def start(s):
    s.dist=dict([(x.id,n) for x in nodes])
    s.dist[s.id]=0
    s.d=n
```

```
    s.first=dict([(x.id,None) for x in nodes])
    s.level=dict([(x.id,-1) for x in s.neigh])
    s.t.start()

  def start1(s):
    s.th.start()

  def getid(s):
    return s.id

  def __repr__(s):
    return str(s.id)

  def check(s):
    s.rmsg=s.buffer.pop(0)
    return s.rmsg

  def check1(s):
    s.rmsg1=s.buffer1.pop(0)
    return s.rmsg1

  def send(s,d,msg):
    d.buffer.append(msg)

  def send1(s,d,msg):
    d.buffer1.append(msg)

  def send2(node,msg):
    node.buffer2.append(msg)

  def add_link(s,z):
    s.neigh.append(z)

  def initiate(s):
    if not s.buffer:
      s.initiated=True
      for node in s.neigh:
        s.send(node,((s.set),s.id))

  def a_test_connectivity(s):
   m=150
   while m:
     m-=1
     sleep(0.01)
```

```
    if not s.buffer2:
     #if s in No:
      s.initiated1=True
      s.reached[s.id]=True
      for node in s.neigh:
        Node.send2(node,(s.id,s.id))
    else:
      if s.buffer2:
         nj,nk=s.buffer2.pop(0)
         if not s.initiated1 :
           s.initiated1=True
           s.reached[s.id]=True
           for node in s.neigh:
             Node.send2(node,(s.id,s.id))
         s.count[nk]+=1
         if not s.reached[nk]:
            s.reached[nk]=True
            s.parent[nk]=nj
            for node in s.neigh:
              if not node.id==nk:
                Node.send2(node,(s.id,nk))
         if s.count[nk]==len(s.neigh):
           if s.parent[nk]:
              if node.id==s.parent[nk]:
                Node.send2(node,(s.id,nk))

def a_compute_distances(s):
  m=80
  while m:
   m-=1
   sleep(0.001)
   s.initiate()
   if s.buffer:
     setj,nj=s.check()
     if not s.initiated:
       s.initiated=True
       for node in s.neigh: s.send(node,((s.set),s.id))
     if s.state<n-1:
       s.level[nj]+=1
       for nk in setj:
         if s.dist[nk]>s.level[nj]+1:
           s.dist[nk]=s.level[nj]+1
           s.d=max(s.dist.values())+choice((0.2,0.4,0.3,0.8,0.9,\
           0.6,0.7,0.5,0.11,0.22,0.33,0.55,0.66,0.77))
```

```
            s.first[nk]=nj
        x=True
        for nb in s.neigh:
          x=x and s.state<=s.level[nb.id]
          if not x: break
        if x:
          s.state+=1
          s.set=Set([])
          for nk in nodes:                                          150
            if s.dist[nk.id]==s.state: s.set.insert(nk.id)
          for nb in s.neigh: s.send(nb,((s.set),s.id))

def extinct_echo_center(s):
  addr=dict([(node.id,node) for node in nodes])
  s.num=len(s.neigh)
  if s.initiator:
    s.caw=s.d
    for q in s.neigh:
      s.send1(q,('tok', s.d, s.id))                                 160
  while s.lrec<s.num:
    sleep(0.01+choice((0.001, 0.002, 0.003)))
    if s.buffer1:
      name,r,qid=s.check1()
      if name=='ldr':
        if s.lrec==0:
          for q in s.neigh:
            s.send1(q,('ldr', r, s.id))
            s.lrec+=1
            s.win=r                                                 170
      if name=='tok':
        if r<s.caw:
          s.caw=r
          s.rec=0
          s.father=qid
          for p in s.neigh:
            if not p.id==qid:
              s.send1(p,('tok', r, s.id))
        if r==s.caw:
          s.rec+=1                                                  180
          if s.rec==s.num:
            if s.caw==s.d:
              for p in s.neigh:
                s.send1(p,('ldr',s.d,s.id))
            else:
```

```
                    #if s.father:
                      s.send1(addr[s.father],('tok',s.caw,s.id))
      if s.win==s.d:
        s.status='Centre'
      else: s.status='remote'                                          190

k=20
nodes=[]
i=0
x=''
x+= 'n0=Node([])\n'
for i in range(k):
  nodes.append(i)
  i+=1
  j=[choice((node,None)) for node in nodes]                            200
  while None in j:
    j.remove(None)
  while j==[]:
    j=[choice((node,None)) for node in nodes]
    while None in j:
      j.remove(None)
  else:
    x+= 'n%d=Node(['%i
    for node in j:
      x+='n%d,'%node                                                   210
    x+='])\n'
    for node in j:
      x+='n%d.add_link('%node
      x+= 'n%d)\n'%i

x+= 'nodes=['
for n in nodes:
  x+= 'n%d,'%(n)
x+= 'n%d]'%k
print x                                                                220
exec(x)


n=len(nodes)

def dot(nodes):
  " make dot file for drawing graph. Use 'dot2ps bf.dot | gv -' "

  f=open('bf.dot','w')
```

```
  f.write('/* automatically generated by a_center2.py */\n')
  f.write('graph G {\n')
  f.write('size="7,8"; ration=compress; \n')
  f.write('node [size="0.01"]\n')
  f.write(' edge [color=blue];\n graph [fontcolor=green,fontsize=18];\n ')
  f.write(' center=true;\n nodesep=0.05;\n margin="0.1,0.1";\n')
  for n in nodes:
    f.write(' %d [shape=ellipse,color=%s,style=filled,label="%d %s"];\n'\
    %(n.id,'yellow',n.id,n.status))
    src=n.id
    for x in n.neigh:
      f.write(' %d--%d [color=%s];\n'%(src,x.id,'red'))
      x.neigh.remove(n) # we don't want to see two links
  f.write('}\n')
  f.close()
  print >>stderr,'bf.dot written - now run "dot -Tps bf.dot | gv -"'


for node in nodes: node.start2()
while 1:
  alive2=0
  for node in nodes: alive2+=node.thh.isAlive()
 # print >>stderr,'%d con threads alive'%alive2
  if alive2==0: break
  sleep(1)
for node in nodes:
  print 'node%d.reached'%node.id, node.reached
sleep(1)

for node in nodes: node.start()
while 1:
  alive=0
  for node in nodes:
    alive+=node.t.isAlive()
 # print >>stderr,'%d path threads alive'%alive
  if alive==0: break
  sleep(1)
print ''
print 'The distance matrix'
for node in nodes:
  print node.id,str(node.dist)
for node in nodes:
  print 'The shortest paht from',node.id,'to all the others is', str(node.d)
print ''
```

```
print 'The table of first neighbor corresponding to the shortes path'
for node in nodes:
  print node.first
sleep(1)

for node in nodes: node.start1()
while 1:                                                              280
  alive1=0
  for node in nodes:
    alive1+=node.th.isAlive()
 # print >>stderr,'%d center threads alive'%alive1
  if alive1==0: break
for node in nodes:
  print node.id,'is', str(node.status)

dot(nodes)
```

Appendix B

Proofs of some theorems mentioned in this report

Proof B.0.1 To avoid a case analysis which are send, receive, or internal events, we represent each event by the uniform notation (c, x, y, d). Here c and d denote the process state before and after the event, x is the collection of messages received in the event, and y is the collection of messages sent in the event. Thus, an internal event (c, d) is denoted (c, O, O, d); a send event (c, m, d) is denoted (c, O, m, d); and a receive event (c, m, d) is denoted $(c, m, \emptyset, d)$. In this notation, event $e = (c, x, y, d)$ of process p is applicable in configuration $\gamma = (c_{p_1}, ..., c_{p_i}, ..., c_{p_N}, M)$ if $c_p = c$ and $x \subseteq M$. In this case

$$e(\gamma) = (c_{p_1}, ..., d, ..., (M\ x) \cup y).$$

Now assume $e_p = (b_p, x_p, y_p, d_p)$ and $e_q = (b_q, x_q, y_q)$ are applicable in

$$\gamma = (..., c_p, ..., c_q, ..., M),$$

that is, $c_p = b_p$, $c_q = b_q$,$x_p \subseteq M$. An important observation is that x_p and x_q are disjoint; the message in x_p (if any) has destination p, while the message in x_q (if any) has destination q.

Write $\gamma_P = e_p(\gamma)$, and note that

$$\gamma_p = (..., d_p, ..., c_q, ..., (M \setminus x_p) \cup y_p).$$

As $x_q \subseteq M$ and $x_q \cap x_p = \emptyset$, it follows that $x_q \subseteq (M \setminus x_p \cup y_p)$, and hence e_q is applicable in γ_p. Write $\gamma_{pq} = e_q(\gamma_p)$, and note that

$$\gamma_{pq} = (..., d_p, ..., d_q, ..., (M \setminus x_p \cup x_q) \setminus x_q) \cup y_q).$$

By a asymmetric argument it can be shown that e_p is applicable in $\gamma_q = e_q(\gamma)$.

Write $\gamma_{pq} = e_p(\gamma_q)$, and note that

$$\gamma_{pq} = (..., d_p, ...d_q, ..., ((M \setminus x_q \cup y_q) \setminus x_p \cup y_p)$$

and hence $\gamma_{pq} = \gamma_{qp}$.

Let e_p and e_q be two events that occur consecutively in an execution, i.e., the execution contains the subsequence

$$..., \gamma, e_p(\gamma), e_q(e_p(\gamma)), ...$$

for some γ. The premise of Theorem applies to these events except in the following two cases.

(1) $p = q$; *or*

(2) e_p is a send event, and e_q is the corresponding receive event.

Proof B.0.2 *Let p_0 be the smallest initiator. The wave initiated by p_0 is joined immediately by every process that receives a message of this wave, and every process completes this wave because there is no wave with smaller identity for which the process would abort the execution of the wave of p_0. Consequently, the wave of p_0 runs to completion, a decision will take place and p_0 becomes leader.*

If p is a non-initiator, no wave with identity p is ever initiated, hence p does not become leader. If $p \neq p_0$ is an initiator, a wave with identity p will be started but a decision in this wave is preceded by a send event (for this wave by p_0) 3.2. As p_0 never executes a send or internal event of the wave with identity p, such a decision does not take place, and p is not elected.

At most n waves are started, and each wave uses at most m messages, which brings the overall complexity to nm.

It is a more delicate question to estimate the time complexity of $Ex(A)$. In many cases it will be of the same order of magnitude as the time complexity of $\mathcal{O}nt$ can be shown (where t is the time complexity of the wave algorithm), because within t time units after initiator p starts its wave, p's wave decides or another wave is started.

Proof B.0.3 As the number of nodes of T exceeds the number of edges by one it suffices to show that T contains no cycle. This follows because, with e_q the first event in q, $qr \in E_T$ implies $e_r \preceq e_q$, and $\preceq$ is a partial order.

Proof B.0.4 As C is a wave there exists an $f \in C_q$ that causally precedes d_p; choose f to be the last event of C_q that precedes d_p. To show that f is a send event, observe that the definition of causality Definition (

Definition B.1 *: 2.20) implies that there exists a sequence (causality chain)*

$$f = e_0, e_1, ..., e_k = d_p$$

such that for each $i < k$, e_i and e_{i+1} are either consecutive events in the same process or a corresponding sent-receive pair. As f is the last event in q that precedes d_p, e_1 occurs in a process different from q, hence f is a send event.

Bibliography

[AF00] A.Wagner and D. A. Fell. The small world inside large metabolic networks. *Technical Report 00-07-041 Santa Fe Institute*, 2000.

[AHT00] Stephen Alstrup, Jacob Holm, and Mikkel Thorup. Maintaining center and median in dynamic trees. In *Scandinavian Workshop on Algorithm Theory*, pages 46–56, 2000.

[Ahu90] M. Ahuja. Flush primitives for asynchronous distributed systems. *Inf. Proc. Lett*, 34:5–12, 1990.

[ALA99] A.-L.Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286:509–512, 1999.

[A.S83] A.Segall. Distributed network protocols. *IEEE Trans. Inform.*, Theory IT-29:23–35, 1983.

[ASBS02] L. A. M. Amaral, A. Scala, M. Barthélémy, and H. E. Stanley. Classes of small world networks. *Proc. Natl. Acad. Sci USA*, 97:11149–11152, 2002.

[Bar96a] Valmir C. Barbosa. An introduction to distributed algorithm. 1996.

[Bar96b] Valmir C. Barbosa. An introduction to distributed algorithms. 1996.

[BG92] Dimitri Bertsekas and Robert Gallager. Data networks (second edition). 1992.

[Bol85] B. Bollobás. Random graphs. *Academic Press, London, UK*, 1985.

[BS03] Stefan Bornholdt and Heinz Georg Schuster. Handbook of graphs and networks, from the genome to the internet. 2003.

[BT96] Dimitri P. Bertsekas and John N. Tsitsiklis. Parallel and distributed computation. 1996.

[CBMT92] B. Charron-Bost, F. Mattern, and Garard Tel. Synchronous and asychronous communication in distributed computations. *Tech. Rep. LTTP*, 92.77, 1992.

[DIM] S. Dolev, A. Israeli, and S. Moran. Uniform self-stabilizing leader election part 1: Complete graph protocols. S. Dolev, A. Israeli and S. Moran, Uniform Self-Stabilizing Leader Election Part 1: Complete Graph Protocols, Technical Report 807, Computer Science Dept., Technion.

[D.Jds] Watts D.J. 1999. *Princeton University Press, Princeton, NJ*, Small Worlds.

[ea] Shlomi Dolev et al. Memory requirements for silent stabilization (extended abstract).

[ER60] P. Erdős and A. Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci., Ser. A*, 5:17–61, 1960.

[GB81] Eli M. Gafni and Dimitri P. Bertsekas. Distributed algorithms for generating loop-free routes in networks with frequently changing topology. *IEEE transactions on communications*, COM-29, NO.1:509–512, 1981.

[GB99] Eli M. Gafni and Dimitri P. Bertsekas. Ad hoc on-demand distance vector (aodv) routing draft-ietf-manet-aodv-04.txt. *Mobile Ad Hoc Network Group*, COM-29, No.1:509–512, 1999.

[Ger91] Gerard.Tel. Fouttolerantie in gedistribueerde algorithm. *Lecture note INF/DOC-91-02, Department Computer science, Utrecht University, The Netherlands, 1991.*, 1991.

[G.L77] G.Lelann. Distributed systems: Towards a formal approach. *In Proc. Information Processing '77, (1977), B. Gilchrist (ed.), North-Holland*, pages 155–160, 1977.

[JM96] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[JSbs] J.M.Montoya and R. V. Solé. 2002. *J. Theor. Biol..in press, Santa Fe Institute Preprint*, pages 01–10–59, Small world patterns in food webs.

[JTA+00] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A.-L. Barabási. The large-scale organization of metabolic networks. *Nature*, 407:651–654, 2000.

[KPBG94] Mehmet Hakan Karaata, Sriram V. Pemmaraju, Steven C. Bruell, and Sukumar Ghosh. Self-stabilizing algorithms for finding centers and medians of trees. In *Symposium on Principles of Distributed Computing*, page 374, 1994.

[Lam78] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communication*, ACM21:558–564, 1978.

[Lyn97] Nancy Ann Lynch. Distributed algorithms, San Francisco, Calif.: Morgan Kaufmann. 1997.

[MPT98] Yoram Moses, Zvi Polunsky, and Ayellet Tal. Algorithm visualization for distributed environments. In *Proceedings IEEE Symposium on Information Visualization 1998*, pages 71–78, 1998.

[MWV00] N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks, 2000.

[NOa] K. Nakano and S. Olariu. Randomized leader election protocols for ad-hoc networks. pages 253–268.

[NOb] Koji Nakano and Stephan Olariu. Uniform leader election protocols for radio networks.

[NO00] Koji Nakano and Stephan Olariu. Randomized leader election protocols in radio networks with no collision detection. In *International Symposium on Algorithms and Computation*, pages 362–373, 2000.

[PC97] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM (3)*, pages 1405–1413, 1997.

[Per97] C. Perkins. Ad-hoc on-demand distance vector routing, 1997.

[RS99] A. Roosta and B. Seyed. Parallel processing and parallel algorithms: theory and computation. 1999.

[Sch] M. Schneider. Flow routing in computer networks.

[SES89] A. Schiper, J. Eggli, and A. Sandoz. A new algorithm to implement causal ordering. *In Proc. 3rd Int. Workshop on distributed algorithms (Nice, 1989), J.-C. Bermond and M. aynal (eds.)*, 392:219–232, 1989.

[SR] Chris Savarse and Jan M. Rabaey. Locationaing in distributed ad-hoc wireless sensor networks.

[Taj77] W. D. Tajibnapis. A correctness proof of a topology information maintenance protocol for a distributed computer network. *Commun. of ACM*, 20(7):477–485, 1977.

[Tan88] A. S. Tanenbaum. Computer networks. page 658, 1988.

[Tel94] Gerard Tel. Introduction to distributed algorithms. 1994.

[WS03] Stefan Wuchty and Peter F. Stadler. Centers of complex networks. *Journal of Theoretical Biology*, 223:45–53, 2003.