1-25-1995

A self-stabilizing distributed maximum flow algorithm

Yan Zhou
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/3142

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
A SELF-STABILIZING DISTRIBUTED MAXIMUM FLOW ALGORITHM

by

Yan Zhou

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science in

Computer Science

Department of Computer Science
University of Nevada, Las Vegas
May 1996
The thesis of Yan Zhou for the degree of Master of Science in Computer Science is approved.

Chairperson, Ajoy Kummar Datta, Ph.D

Examinining Committee Member, Lawrence L. Larmore, Ph.D

Examinining Committee Member, John T. Minor, Ph.D

Graduate Faculty Representative, Ashok Iyer, Ph.D

Interim Dean of the Graduate College, Cheryl L. Bowles, Ed.D

University of Nevada, Las Vegas
May 1996
ABSTRACT

This thesis presents a self-stabilizing distributed maximum flow algorithm for a network $G = (V, E)$, where $V$ is a set of nodes in the network and $E$ is a set of edges in the network. The algorithm has two phases: reset phase and preflow-push phase. Fault-tolerance is achieved by using a self-stabilizing paradigm that uses non-masking fault-tolerance embedded repetitions within the algorithm. Two techniques are used in the algorithm. Counter flushing is used to synchronize the network; both local checking and local correction are used to compute the maximum flow of the network. The algorithm handles catastrophic faults by weeding out false information in the network. A network can start with any arbitrary global state and will recover to a legal global state in finite number of steps. Lastly, the network guarantees to restore the legal configuration from any catastrophic faults.
LIST OF FIGURES

Figure 1  Protocol for Source Node s ................................. 37
Figure 2  Protocol for Nodes Other Than s and t .................. 38
Figure 3  Protocol for Sink Node t ................................. 39
Figure 4  The Variables Used in Each Node .................. 40
Figure 5  All Support Functions and Preflow_push Code ........ 41
ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Ajoy Kumma Datta for his constant support, helpful advice and encouragement throughout the course of this thesis. Special thanks to Dr. John Minor for his careful review. I would also like to thank Dr. Lawrance Larmore, Dr. John Minor, and Dr. Ashok Iyer for serving on my committee.

Y. Zhou
Chapter 1

INTRODUCTION

One of the major issues that make designing network protocols complicated is fault-tolerance. There are two general fault models: Byzantine fault model and self-stabilization model. The self-stabilization model allows an arbitrary number of faults that stop (catastrophic faults) while the Byzantine model allows a limited number of faults that continue. Catastrophic faults include common failures, such as node and link crashes, memory corruption, or malfunctioning devices sending out incorrect messages.

The term self-stabilization was originally introduced by Dijkstra[12] to distinguish any system having the property that, regardless of the current state, the system guarantees to recover to a legal global state in a finite number of steps. Once in a legal state it will remain thereafter, until subsequent catastrophic faults occur. Self-stabilization is attractive for networks for the following three reasons: catastrophic faults do occur from time to time; manual intervention has a high cost; for a distributed program, an initial state seems to be an artificial concept.

1.1 CONCEPTS AND NOTATIONS

A distributed system consists of a set of processors, $P_1, P_2, ..., P_n$, which are interconnected with communication channels, $(P_i, P_j)$. The system is represented by a graph $G = (V, E)$, where $V$ is the set of processors and $E$ is the set of connecting channels between any two neighboring processors. The terms process, processor, and
node are interchangeable, and also the terms channel, link, and edge are interchangeable throughout this paper.

A network protocol consists of a program for each network node. A program is a set of variables and a finite set of actions. Each variable has a predefined nonempty domain. Each action has the form: \(<\text{guard}> \rightarrow \langle \text{statement} \rangle\)

A guard of a process P is one of the following: a local guard of P, or a receiving guard of P. A local guard of P is a Boolean expression over its own variables and the variables of its neighbors. A receiving guard of process P is of the following form:

\text{recv} \langle \text{message} \rangle \text{ from } \langle \text{process Q} \rangle

A statement of P is one of six forms: skip, assignment, sending, sequence, selection, and iteration.

A skip statement of process P is of the following form: \text{skip}

This statement is executed by doing nothing.

An assignment statement of process P is of the following form:

\text{x.0, ..., x.k := E.0, ..., E.k}

The assignment statement is executed by first computing the current values of all the E.i’s then assigning the computed value of each E.i to the corresponding variable x.i.

A sending statement of process P is of the following form:

\text{SEND}_{p, q} (\text{message})

A sequence statement of process P is of the following form:

<\text{statement of P}>; ...; <\text{statement of P}>
This statement is executed by first executing the first statement of P then executing the second statement of P.

A selection statement of process P is of the following form:

\[
\text{if } gd.0 \rightarrow sm.0 \sqcup \ldots \sqcup gd.k \rightarrow sm.k \text{ fi}
\]

Each gd.i is a local guard of P such that the disjunction of the gd.i's is true for any assignment of values to the variables of P. Each sm.i is a statement of P. The selection statement is executed by first computing the current Boolean values of the gd.i's then selecting arbitrarily one gd.i whose value is true and executing its corresponding sm.i.

An iteration statement of process P is of the following form:

\[
\text{do } gd \rightarrow sm \text{ od}
\]

In this statement, gd is a local guard of P and sm is a statement of P. The iteration statement is executed by repeatedly computing the current Boolean value of gd then executing statement sm, when gd is true. Execution of the iteration statement terminates when gd is false. We require that, for each assignment of values to the constants, inputs, and variables of P, there is an integer k such that the execution of the iteration statement is guaranteed to terminate within k iterations.

A state of a process is defined by a value for each variable in the process. A global state is the Cartesian product of the states of each process in the system. An action whose guard is true at some system state is said to enabled, and a process with an enabled rule is said to be privileged. When a process is privileged, it will within a finite amount of time make a move, which changes its local state and global state.
1.2 General Techniques

There are several general techniques for self-stabilization that have been studied. In this section we will briefly discuss all those techniques.

1.2.1 Global Checking and Global Correction

This is the first general technique for self-stabilization [24]. The basic idea is to add a leader node, periodically do a snapshot of the network, and reset the network if a global inconsistency is detected.

The advantage of this technique is its generality. Since the general transformation is expensive, the search for techniques that are less general and more efficient has continued.

1.2.2 Local Checking and Local Correction

Local checking is to detect inconsistent global states by checking the states of each subsystem and local correction is to correct each subsystem independently.

This technique is much more efficient. To use this technique in a network protocol, the protocol must be locally checkable and locally correctable. A protocol is said to be locally checkable if the conjunction of legal states of each process leads to a legal global state. A protocol is locally correctable if when each subsystem is corrected independently, the global state will eventually be legal. So its not hard to see that if a protocol is locally checkable, its dependency relation must be acyclic. Local checking
and correction can be used to design self-stabilizing protocols for mutual exclusion, the end-to-end problem, etc.

However, not every locally checkable protocol can be locally correctable (e.g., spanning tree construction and topology update[33]). So another general method, local checking and global correction, is suggested.

1.2.3 Local Checking and Global Correction

The ideas of local checking and global correction are: illegal global states are detected by local checking mechanism, and a global correction action (called “reset”) is used to recover from faults.

For the following reasons, this method is the right balance in many practical situations[8]. First, global checking mechanisms such as the self-stabilizing snapshot incur unnecessary large overhead in terms of time, space and communication, since networks are fairly failure-free. Local checking detects faults quickly, and it can be done with only a small increase in communication cost. Secondly, as mentioned above, there are many protocols that are locally checkable but not locally correctable. Finally, resetting an entire network is not as drastic and inefficient as it seems to be. This is due to the fact that the stabilization time of best reset protocols is proportional to cross-network latency, which is the time it takes for many protocols to compute their results.

Unfortunately, there are some protocols that are neither locally checkable nor local correctable. Next, we will describe another general technique, counter flushing, which is applicable to those protocols.
1.2.4 Counter Flushing

The counter flushing method was introduced by Varghese[33]. By attaching a simple counter to the state of every node and to every message, nodes in the network will only accept messages with a counter (different to their own counter) from their parents, and will only accept messages with a counter (the same as their own counter) from their children. Nodes will periodically send messages to their children or parents depending on the situation. In this way, eventually only the correct information is being passed around the network. The protocol will begin to work correctly regardless of the initial messages and node states.

This technique appears to be applicable to several total algorithms (i.e., algorithms that involve the cooperation of all nodes in the network), such as token passing[12], propagation of information with feedback, deadlock detection, network resets, and non-blocking network snapshots.

In some cases, both counter flushing and local checking and correction techniques are applicable. However, the method of local checking requires a fairly tedious enumeration of the protocol invariants which need to be checked. The addition of local checking also has a fair amount of complexity. Also, taking correct snapshots of local states requires some careful synchronization which makes actual implementations difficult. By contrast, the modifications required to implement counter flushing are extremely simple. So when both methods are applicable, the counter flushing method is preferred.
1.2.5 Closure and Convergence

Closure and convergence is another general technique introduced by Arora and Gouda[4].

In this method, programs are a set of legal actions $p$ and a class of fault actions $F$. There are both a set of legal states $R$ and a large set of states $Q$ which are represented by a state predicate $S$(called the invariant) and another state predicate $T$(called the fault-span), respectively. $p$, $F$, $S$, $T$ must satisfy following conditions:

a) Inclusion: $T \subseteq S$

b) closure: $S$ is closed in $p$; $T$ is closed in $p$ and $F$.

c) Convergence: $T$ converges to $S$ in $p$, i.e., when actions in $F$ stop executing, subsequent execution of actions in $p$ alone eventually yields a state where $S$ holds, from which point the program resumes its intended execution until the next fault action occurs.

In the method, four design steps are performed:

1. Design of the fault-span $T$: A state predicate $T$ is constructed that is weak enough so that the fault actions preserve it.

2. Design of the invariant $S$: A state predicate $S$ is constructed that is strong enough to meet the safety properties of the problem specification.

3. Design of program actions that achieve nonmasking tolerance: Program actions are constructed that ensure $T$ converges to $S$.

4. Design of program actions that satisfy the problem specification: Program actions are constructed to satisfy the problem specification in all computations that start from states where $S$ holds. Each of these actions is verified to preserve $S$ as well as $T$. 
Chapter 2

MAXIMUM FLOW ALGORITHM

Before going any further, we assume that an underlying breadth-first tree protocol exists to construct Breadth-first search tree (BFT) for the network [22]. The reason for using BFT, other than a general spanning tree, is that we want the lowest height of the spanning tree. In the following algorithm, this underlying spanning tree is only used for sending and receiving messages to synchronize the network, and is not involved in computing the maximum flow. In addition, the synchronizing time depends on the height of the tree so BFT is chosen for minimizing the height of the tree. Because the underlying BFT protocol only stabilizes the children, (children’s id set of node u) and parent, (id of the parent node of node u in BFT) variables, the maximum flow protocol uses these two variables to synchronize the network. So these two protocols have disjoint written sets and can stabilize independently.

In this thesis we use the preflow-push method to compute the maximum-flow. The self-stabilization version of preflow-push protocol is based on Goldberg’s “generic” maximum-flow algorithm, which has a simple implementation running in $O(|V|^2|E|)$ time that is better than the $O(|V||E|^2)$ bound of Edmonds-Karp’s algorithm. The protocol has a stabilizing termination property: that is, after the network is initiated, its preflow-push computation will terminate in finite number of steps no matter what state the algorithm started with. When its source node s detects the termination it will issue another wave of
initialization. Because of this, the protocol can reset and re-compute periodically. If the
initial state is incorrect or network has been changed since last initialization, the reset
phase will correctly reset the network, and the future execution of preflow-push phase
will give the correct maximum flow.

2.1 Maximum Flow Problem

The maximum-flow problem is the simplest problem concerning flow networks,
and also it is a very important problem. Not only can this problem be solved by efficient
algorithms, also the basic techniques used by these algorithms can be adapted to solve
other network-flow problems. A graph-theoretic definition of flow networks will be
given next.

A flow network $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has
a nonnegative capacity $c(u, v) \geq 0$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We
distinguish two vertices in a flow network: a source $s$ and a sink $t$. For convenience, we
assume that every vertex lies on some path from the source to the sink. That is, for every
vertex $v \in V$, there is a path $s \rightarrow \ldots \rightarrow v \rightarrow \ldots \rightarrow t$. The graph is therefore connected, and
$|E| \geq |V| - 1$.

A flow in $G$ is a real-valued function $f : V \times V \rightarrow \mathbb{R}$ that satisfies the following
three properties:

**Capacity constraint:** $f(u, v) \leq c(u, v)$, for all $u, v \in V$.

**Skew symmetry:** $f(u, v) = - f(v, u)$, for all $u, v \in V$.

**Flow conservation:** $\sum_{v \in V} f(u, v) = 0$, for all $u \in V - \{s, t\}$. 
The quantity $f(u, v)$, which can be positive or negative, is called the net flow from vertex $u$ to vertex $v$. The value of a flow $f$ is defined as $|f| = \sum_{v \in V} f(s, v)$, that is, the total net flow out of the source. In the maximum-flow problem, we are given a flow network $G$ with source $s$ and sink $t$, and we wish to find a flow of maximum value from $s$ to $t$.

Intuitively, given a flow network and a flow, the residual network consists of edges that can admit more net flow. More formally, suppose that we have a flow network $G = (V, E)$ with source $s$ and sink $t$. Let $f$ be a flow in $G$, and consider a pair of vertices $u, v \in V$. The amount of additional net flow we can push from $u$ to $v$ before exceeding the capacity $c(u, v)$ is the residual capacity of $(u, v)$, given by $c_f(u, v) = c(u, v) - f(u, v)$. The residual network of $G$ induced by $f$ is $G_f = (V, E_f)$, where $E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$. An augmenting path $p$ is a simple path from $s$ to $t$ in the residual network $G_f$.

A flow $f$ of a network $G = (V, E)$ is a maximum flow if and only if $f$ satisfies the following:

1. $f(u, v) \leq c(u, v)$, for every edge $(u, v) \in E$.
2. $\sum_v f(u, v) - \sum_u f(v, u) = 0$, for all $u \in V$.
3. There is no augmenting path in the $G_f$.

There are two basic methods to solve the maximum-flow problem: Ford-Fulkerson method and Preflow-push method.

All of the algorithms based on Ford-Fulkerson method are iterative. At each iteration, the flow value is increased by finding an augmenting path $p$, along which more flow is pushed. This process is repeated until no augmenting path can be found. So those
algorithms start with an assignment \( f \) that satisfies condition (1) and (2) and terminate with an \( f \) satisfying (3), always maintaining (1) and (2) at each step of the algorithm.

Another class of algorithms are based on preflow-push method. Those algorithms are more general, more powerful, and more flexible than augmenting path algorithms. The best preflow-push algorithms currently have better performance than the best augmenting path algorithms in theory as well as in practice.

The generic preflow-push algorithm has a somewhat different intuition. For better understanding, we consider a flow network \( G = (V, E) \) to be a system of interconnected pipes of given capacities. Directed edges correspond to pipes. Vertices are pipe junctions, and have two interesting properties. First, to accommodate excess flow, each vertex has an outflow pipe leading to an arbitrarily large reservoir that can accumulate fluid. Second, each vertex, its reservoir, and all its pipe connections are on a platform whose height increases as the algorithm progresses.

Vertex heights determine how flow is pushed: we only push flow downhill, that is, from a higher vertex to a lower vertex. There may be positive net flow from a lower vertex to a higher vertex, but operations that push flow always push it downhill. The height of the source is fixed at \( |V| \), and the height of the sink is fixed at 0. All other vertex heights start at 0 and increase with time. The algorithm first sends as much flow as possible downhill from the source toward the sink. The amount it sends is exactly enough to fill each outgoing pipe from the source to capacity; that is, it sends the capacity of the cut \((s, V - s)\). When flow first enters an intermediate vertex, it collects in the vertex's reservoir. From there, it is eventually pushed downhill.
It may eventually happen that the only pipes that leave a vertex \( u \) and are not already saturated with flow connect to vertices that are on the same level as \( u \) or are uphill from \( u \). In this case, to rid an overflowing vertex \( u \) of its excess flow, we must increase its height -- an operation called “lifting” vertex \( u \). Its height is increased to one unit more than the height of the lowest of its neighbors to which it has an unsaturated pipe. After a vertex is lifted, therefore, there is at least one outgoing pipe through which more flow can be pushed.

Eventually, all the flow that can possibly get through to the sink has arrived there. No more can arrive, because the pipes obey the capacity constraints; the amount of flow across any cut is still limited by the capacity of the cut. To make the preflow a “legal” flow, the algorithm then sends the excess collected in the reservoirs of overflowing vertices back to the source. (Shipping the excess back to the source is actually accomplished by canceling the flows that cause the excess.) As we shall see, once all the reservoirs have been emptied, the preflow is not only a “legal” flow, it is also a maximum flow.

A preflow-push algorithm starts with an assignment \( f \) that satisfies conditions (1) and (3) and terminates with an \( f \) that satisfies (2), always maintaining (1) and (3) at each step in the algorithm.

The maximum flow algorithm due to Goldberg starts with the preflow \( f \) that is equal to the capacity \( c(s, v) \) on every edge \((s, v)\) directed away from \( s \), and zero on all other edges, and with some initial valid height. The simplest choice of a valid initial
height is $h(s) = |V|$, and $h(v) = 0$ for all $v \neq s$. The algorithm then repetitively performs two basic operations, push and lift, as described in the following:

**Push (v, w)**

*Applicability.* $v$ is active (i.e., excess flow $e(v) > 0$), $(v, w) \in E_r$ and $h(v) = li(w) + 1$.

*Action.* Set $\delta = \min\{e(v), c(v, w)\}$ and do the following:

1. Increase $f(v, w)$ by $\delta$ and decrease $f(w, v)$ by $\delta$.
2. Decrease $e(v)$ by $\delta$ and increase $e(w)$ by $\delta$.
3. Update the residual network $G_f$.

**Lift(v)**

*Applicability.* $v$ is active, and for every $(v, w) \in E_r$, $h(v) \leq h(w)$.

*Action.* Set $h(v) = \min\{h(v), h(w) + 1\}$.

A push from $v$ to $w$ is a *saturating* push if $c_t(v, w) = 0$ after the push; otherwise it is a *nonsaturating* push.

Next is the generic maximum flow algorithm due to Goldberg:

**step 1:** (initialize preflow $f$ and valid height $h$) Set

1. $f(s, v) = c(s, v)$, for every $(s, v) \in E$.
2. $f(v, w) = 0$, for every $(v, w) \in E$ with $v \neq s$.
3. $h(s) = |V|$, and $h(v) = 0$ for all $v \neq s$.

**step 2:** If there is no active vertex, stop. ($f$ is a maximum flow.)

**step 3:** Select an active vertex and apply a basic operation. Go to step 2.

Note that in above algorithm the basic operations can be applied in any order.
2.2 Algorithm

In our protocol, there are two levels of hierarchy. At the top level, a reset protocol periodically resets the whole network correctly, after a reset, at the bottom level, a preflow-push protocol computes the maximum flow of the network. Once source node s detects that the preflow-push protocol has terminated, s resets the network again, and starts a new wave. The source node s guarantees to alternate between reset phase and preflow-push phase since both phases guarantee to terminate. The reset phase guarantees to terminate because it uses a stabilizing reset, and the preflow-push phase guarantees to terminate because of its stabilizing termination property. The alternation is accomplished by a variable \textit{turn}, at the root node s. This variable \textit{turn} can take only one of the two values: \textit{reset} and \textit{preflow-push}. When the \textit{turn} is \textit{reset} and \textit{reset\_finished}(s) is true, s sets \textit{turn} variable to \textit{preflow-push}. When \textit{turn} is \textit{preflow-push} and preflow-push protocol terminates (i.e., preflow-finished(s) is true) or when node s receives a \textit{reset\_request} message with the current counter, the source node s sets \textit{turn}, variable to \textit{reset} and a new wave of reset starts.

After a successful reset phase, all nodes in the network are initialized correctly for the preflow-push protocol. Thus this phase will work correctly and upon terminating it will provide the correct maximum flow result. In the preflow-push phase of the protocol, we keep two sets of variables. One set of variables \( f(u,v) \) is used to hold the netflow from node \( u \) to node \( v \), and is updated during the computation of maximum flow. The other set of variables \( f_{\text{final}}(u,v) \) is the output of the flow network. If a new reset wave is issued after a successful maximum flow computation, \( f(u,v) \) is copied to
 During the LOCAL_RESET procedure. Otherwise, reset wave is issued upon receiving a reset_request message. In this case, the preflow-push phase was not complete, and $f(u,v)$ will not be copied to $f_{\text{final}}(u,v)$ during the LOCAL_RESET procedure. This is controlled by a Boolean variable $copy_flag$.

In the protocol, there are two techniques being used. The counter flushing method is used to synchronize the whole network for a reset by using messages; local checking and local correction methods are used to compute maximum flow of the network by using the preflow-push approach.

In the maximum flow computation, of all nodes involved, there are two special nodes: source node $s$ and sink node $t$. They have to be treated differently and require different codes.

Code for process $s$ is shown in figure 1. There are seven rules for it.

Rule 1: If process $s$ is in preflow_push phase and the phase is not finished, then **PRE_FLOW_PUSH($s$)** protocol is executed.

Rule 2: If process $s$ is in preflow-push phase and the phase is finished (i.e. the current wave of maximum flow computation is finished), $s$ issues a new wave of reset by executing the **REQUEST_RESET(copy_flag,$s$)** procedure. In that procedure, variable $\text{turn}$ will be set to $\text{reset}$.

Rule 3: If process $s$ is in reset phase and function **FINISHED($s$)** returns true, then the reset phase is finished. Variable $\text{turn}$ is set to preflow_push. Preflow-push phase starts at this moment.
The previous three rules are used to control the network alternating between reset phase and preflow-push phase.

Rule 4: Process s periodically sends a message tuple \((turn, copy\_flag, c_s)\) to all its children.

Rule 5: Upon receiving a message tuple \((turn, copy\_flag, counter)\) from its child \(v\), process s will set \(reset\_finish\_token\_expected, (v)\) to \(false\) if \(c_s\) equals to \(counter\) and both \(turn_s\) and \(turn\) are reset, this is due to the fact that s knows that all nodes in the subtree(rooted in node \(v\)) have finished resetting their local variables.

Rules 4 and 5 are used to synchronize the network and flush out the old information.

Rule 6: When process s receives a message tuple \((flow, counter)\) from its child process \(v\), it sets its variable \(flow\) to \(flow_v\) if \(counter\) is equal to \(c_s\). The variable \(flow\) is used to check whether the preflow-push phase is finished or not. According to the rules for any processes other than s, we can see that the message tuple \((flow, counter)\) is sent only at the time that the network is in preflow-push phase. If the network is not in preflow-push phase, then the received message \((flow, counter)\) is old information, and s will ignore it. That is why \('turn = "preflow\_push"' is part of its guard in rule 6.

Rule 7: This rule is used when process s receives a reset_request message. If the request is not out-of-date, then s will do the \texttt{REQUEST\_RESET(0)} procedure.

Code for processes other than s and t is shown in figure 2. There are a total five rules for it.
Rule 1: If variable $\text{turn}$ is $\text{preflow\_push}$, the \texttt{PRE\_FLOW\_PUSH}(u) protocol is executed.

Rule 2: Process $u$ periodically sends a message tuple $(\text{turn}_u, \text{copy\_flag}_u, c_u)$ to all its children. If it is in $\text{reset}$ phase and all its children have finished their local resets, then $u$ sends this information up to its parent.

Rule 3: When process $u$ receives a message tuple $(\text{turn}, \text{copy\_flag}, \text{counter})$ from process $v$, it executes $\text{T\_RECEIVE}_{u,v}(\text{turn}, \text{copy\_flag}, \text{counter})$ procedure. This procedure will be described later.

Rule 4: When process $u$ receives a message tuple $(\text{flow}_v, \text{counter})$ from its child process $v$, it executes $\text{F\_RECEIVE}_{u,v}(\text{flow}_v, \text{counter})$ procedure. This procedure is simple. If $c_v$ equals to the $\text{counter}$, then $u$ sends flow information to its parent. Otherwise, the flow information is old and will be simply ignored.

Rule 5: When process $u$ receives a $\text{reset\_request}$ message from its child, if this is new information, then it forwards this information to its parent.

Code for sink node $t$ is almost the same as the code in figure 2. There are two differences that need to be mentioned. The first difference is that in rule 1, after executing $\text{PRE\_FLOW\_PUSH}(t)$, process $t$ calculates its netflow and then sends flow information to its parent. Node $s$ can determine whether or not the preflow-push phase is finished when it receives this flow information. The second difference is that for node $t$, rule 4 is not necessary, because $t$ is the source of the flow information message. It will never receive flow information from any other nodes. Thus, the rule 4 in this protocol is
correspondent to rule 5 in previous code for node u. Code for sink node is given in figure 3.

We now describe each support function. These functions are shown in figure 4, along with description of variables and data structures used by the protocol.

One function is the \textsc{Choose}(MAX, c) function. This function chooses a new counter value c that is a positive integer less than or equal to MAX. This function can be implemented in three different ways as discussed in [20]. We assume that \textsc{Choose} is implemented by a simple increment such that if current value for the counter at s is \( c_s \), then \textsc{Choose} will take the new value \( c_s = c_s + 1 \mod MAX \). This assumption is based on the fact that it does not decrease the time complexity of the overall algorithm, and it can simplify the proof.

\textsc{Finished}(u) is a Boolean function. Each node u uses the function to determine whether or not it is expecting an input from its children. Node u will expect an input from its children whenever it has forwarded new reset information from the root to its children. \textsc{Finished}(u) is \textit{true} when and only when node u has received a response from all of its children during the current reset wave.

\textsc{Preflow-Finished}(s), another Boolean function, is used for source node s only. If node s received a flow information that was equal to its netflow, the preflow_push phase of the network ended up with a correct maximum flow result. Otherwise, if the flow information received was greater than its netflow, which was the case that an incorrect initiation was conducted, then the preflow-push phase was aborted. In the former case variable \textit{copy-flag}_s is set to 1, while in the latter case it is
set to 0, because for the former case the correct result needs to be copied into variable $f_{final}$ while incorrect result should be ignored for the latter case. In both cases, the function returns `true` to indicate finishing of the preflow-push phase. Otherwise, it returns `false`.

`REQUEST_RESET(copy_flag)` is a procedure used only by source node $s$. It first locally resets node $s$, then sets variable `turn` of node $s$ to `reset` and chooses a new counter. By means of `SEND_v,\langle turn_v, copy_flag_v, c_v \rangle` in rule 4 of process $s$, node $s$ issues a new wave of reset. Finally, the procedure sets Boolean variable `reset_finish_token_expected(v)` for every child $k$ of $s$ to `true` and variable `flow` to 0. This bunch of `reset_finish_token_expected(v)` is used in function `FINISHED(s)` to decide whether or not this wave of reset is finished, while variable `flow` is used in function `PREFLOW_FINISHED(s)` to decide if the preflow-push phase is finished.

`T_RECEIVE_v,\langle turn, copy_flag, counter \rangle` is a procedure that is used by node $s$ only. Node $s$ receives a message tuple `(turn, copy_flag, counter)`. If the message has the same counter with $c_v$ and variable `turn` is `reset`, then the subtree rooted at node $v$ must have finished resetting. Therefore, the procedure assigns `reset_finish_token_expected(v)` to `false`.

`T_RECEIVE_u,\langle turn, copy_flag, counter \rangle` is a procedure that is used by every node $u$ except source node $s$. There are two cases that need to be considered. The first, a message is received from $u$’s parent. There are three related subcases: 1) `Counter` is fresh and `turn` is `reset`, then variable `turn_u` is set to `turn`. This is a new reset phase, so the procedure calls `LOCAL_RESET(u)` and sets `reset_finish_token_expected_u(v)` to `true`
for every child k of u. 2) \textit{Counter} is the same as \( c_u \). \textit{turn} is \textit{preflow-push}, then \( \text{turn}_u \) is set to \textit{preflow-push} no matter what \( \text{turn}_u \) is. Node u enters or remains in preflow-push phase. 3) \textit{Counter} is different from \( c_u \), \textit{turn} is \textit{preflow-push}. This happens only when the system is still recovering from some illegal global state (Because it will never happen if the preflow-push phase is following a complete reset phase). The system was not synchronized when this preflow-push phase was issued from source node s. So the procedure sets \( c_u \) to \textit{counter} to make the node u the same counter value as its parent's. Node u then sends \textit{reset-request} message to its parent. When its parent receives this message it will forward the message to its own parent. With this process going on, the source node s will finally receive this message and start a new reset phase. The second case: if the message is from its child v, both \( \text{turn}_u \) and \( \text{turn} \) are \textit{reset}, and \( c_u \) equals to \textit{counter}. In this case the subtree rooted at node v finishes resetting. So Boolean variable \texttt{reset\_finish\_token\_expected}_u(v) is set to \texttt{false}.

Three \texttt{LOCAL\_RESET} functions (\texttt{LOCAL\_RESET}(s), \texttt{LOCAL\_RESET}(t), and \texttt{LOCAL\_RESET}(u)) will be explained next. According to the initiation of this preflow-push method, the height of node s is the number of nodes of the network. The excess flow into node s is always 0. Because each edge leaving the source node s is filled to capacity, so the netflow of each edge \( f(s, v) = c(s, v) \). The residual capacity of edge \( (s, v) \) is : \( c_f(s, v) = c(s, v) - f(s, v) = 0 \). For sink node t, its height is fixed at 0. By definition, the excess flow into node t is also 0. Because each edge \( (t, v) \) is empty, so netflow \( f(t, v) = 0 \). Since no net flow is filled in edge \( (t, v) \), the residual capacity \( c_t(t, v) \) of edge \( (t, v) \) remains the same as capacity of edge \( (t, v) \) at this point. If t is a neighbor of
source node s, the net flow and residual capacity of edge (t, s) need to be considered separately. According to the skew symmetry property, $f(u, v) = -f(v, u)$ leads to $f(t, s) = -f(s, t) = -c(s, t)$. The residual capacity $c_f(t, s) = c(t, s) - f(t, s) = c(t, s) + c(s, t)$. For every other node $u$, the height is 0. If $u$ is adjacent to source node $s$, the excess flow into node $u$ is $c(s, u)$. Otherwise, $e(u)$ is 0. For every edge $(u, v)$, the net flow $f(u, v) = 0$, and the residual capacity $c_f(u, v) = c(u, v)$. If $u$ is a neighbor of source node $s$, then $f(u, s) = -c(s, u)$, the residual capacity $c_f(u, s) = c(u, s) - f(u, s) = c(u, s) + c(s, u)$.

**COPY_NETFLOW(u)** is simply a function which copies each netflow $f(u, v)$ of last maximum flow computation back to variable $f_{\text{final}}(u, v)$, so that other application program can use it.

**PRE_FLOW_PUSH** protocol was written by using local checking and local correcting techniques. Each node has its own protocol: **PRE_FLOW_PUSH(s)** for source node $s$, **PRE_FLOW_PUSH(t)** for sink node $t$, and **PRE_FLOW_PUSH(u)** for every other node $u$. In the preflow_push method, there are two legal actions: **push** and **lift**. Push action can be applied from node $u$ to node $v$ when following three conditions are satisfied:

1 ) node $u$ is an overflowing vertex. i.e. excess flow $e_u > 0$.

2 ) residual capacity of edge $(u, v)$ is more than 0. i.e. $c(u, v) > 0$.

3 ) height of node $u$ is equal to one plus height of node $v$. i.e. $h_u = h_v + 1$.

Lift action can be applied to a node $u$ when following two conditions are true:

1 ) node $u$ is an overflowing vertex.

2 ) $h_u \leq h_v$ for every $v \in N_u$ and $c_f(u, v) > 0$.
**PRE_FLOW_PUSH(u)** is a protocol that has six rules. One rule for push action, one for lift action, and the other four for correction.

Rule 1: If there is such a node u’s neighbor node v that above three conditions are satisfied, the push action can be applied in fault-free situation. \( \forall v(f(u, v) = -f(v, u)) \) is used to make sure that node u collects all flows being pushed towards it from all its neighbors after u’s last push operation. Since we want the protocol to be more robust, three other conditions are added to avoid having push action executed under the situation that \( e_u, f(u, v) \) or \( c(u, v) \) is wrong. \[ e_u = \{ \sum f(u, v) \mid v \in N_u \}, f(u, v) = c(u, v) - c_f(u, v), \text{ and } c_f(u, v) + c_f(v, u) = c(u, v) + c(v, u) \] (in the last two propositional expressions, node v is the node that the flow is going to be pushed to.) are three extra conditions. They are served together to protect node u from doing push operations under wrong situations. This is very important, just think about the following situation: if \( e_u \) is supposed be 0, but because of memory corruption, somehow \( e_u \) becomes a large enough positive number x, then the whole flow in the network is now \( \{ \sum c(s, v) \mid v \in N_s \} + x \) instead of \( \{ \sum c(s, v) \mid v \in N_s \} \), and the overflow might never be able to pushed back to s. This makes the preflow_push phase never terminate. If all seven conditions are satisfied, then push action will be executed. It first calculates the units of flow to be pushed and temporarily stores them in variable \( d_i(u, v) \). Then it will update the excess flow \( e_u \), net flow \( f(u, v) \), and residual capacity \( c(u, v) \). Because a node can read/write its own local variables, and can only read its neighbors’ variables, so changes of node v’s \( e_v, f(v, u) \), and \( c(v, u) \) caused by this push operation will be taken care of by the next rule.
Rule 2: If a neighbor node v of node u pushed $d(v, u)$ to node u, and node u did not change its local variable $e_u$, $f(u, v)$ and $c(u, v)$, then skew symmetry property has been violated $f(u, v) + f(v, u) = d(v, u)$. If node u and node v pushed both ways and both nodes don’t change their variables, then $f(u, v) + f(v, u) = d(v, u) + d(u, v)$. In these two cases node u has to collect the flow pushed from node v to itself. So $f(u, v) = f(u, v) - d(v, u); c(u, v) = c(u, v) + d(v, u); e_u = e_u + d(v, u)$. $f(u, v) = c(u, v) - c(u, v)$ and $f(v, u) = c(v, u) - c(v, u)$ are used to make sure that $f(u, v)$ and $f(v, u)$ are not changed by any fault.

Rule 3: The lift action is enabled when those two conditions are met. The other three conditions are added, for the same reason that was mentioned in rule 1. When all five conditions are true, the height of node u is lifted to the one more than the minimum of the $h_v$, for every $v$ that $c(v, u) > 0$.

Rule 4: If the excess flow of node u is not equal to the sum of the netflow entering this node as it is supposed to be, then the excess flow is assigned to the sum of the netflow entering this node.

Rule 5: If the netflow from node u to node v is not equal to capacity minus residual capacity as it is defined, then it is assigned to what it is defined.

If faults affect $e_u$, $f(u, v)$, or both, rule 4 and rule 5 together with rule 1 and rule 3’s three extra conditions will recover them from the faults perfectly. If faults affect residual capacity, the situation is much worse. When some faults, such as memory corruption, changes $c(u, v)$, which makes $c(u, v) + c(v, u) \neq c(u, v) + c(v,u)$. There is no way to know if either one residual capacity is changed, or both are changed and how
they are changed. It is impossible to make a correct guess. Any wrong guess will result in an incorrect maximum flow or even cause the preflow-push phase never to terminate. Because of these reasons, rule 6 is designed to handle this situation.

Rule 6: There is no point in making any guess. since the guess will be wrong anyway, this rule simply sends a reset_request message along with its counter up to its parent. According to overall protocol for node u, s and t, when any node u receives this message, it will forward this message to its parent. Finally the source node s will get this request and reset the whole network. Once the guard for rule 6 is true, it will remain true. This rule will periodically send out this request message until a new reset wave is issued by node s.

Rule 7: According to preflow-push method. if $e_u > 0$ and $c_e(u, v) > 0$ then $h_u$ is always less than or equal to $h_v + 1$. If $h_u > h_v + 1$, there must be something wrong. To recover it, we can simply make $h_u = h_v + 1$.

Pre_flow_push(s) and Pre_flow_push(t) are protocols for node s and node t, respectively.

Because both node s and node t have their heights fixed and their excess flows are always zero, there is no need to take push and lift actions. Both of them have four correction rules. Those correction rules are similar to those in Pre_flow_push(u). The only differences are: 1) always set $e_s$ and $e_t$ to zero when they are not equal to zero. 2) always make $h_s = |V|$ and $h_t = 0$ when they are not equal to those values.
2.3 Correctness Reasoning

Lemma 1 Any counter value c sent by the root node will reach and be accepted by all nodes in the tree within $O(h)$ time, where $h$ is the height of the tree.

Proof: By induction on the distance from the source. The number of links between a particular node $u$ and the source node $s$ is considered as the distance from the source node $s$ to node $u$.

Basis: For the source node $s$, its distance to itself is zero. It is trivial to see that the root has the counter value $c$ since that value is created at the source node by using the $\text{CHOOSE}(\text{MAX}, c)$

Induction: By the induction hypothesis, assuming that all nodes at a distance $δ-1$ from the source will have received and accepted the value $c$. It must be shown that all nodes at a distance $δ$ receive the counter value $c$. All nodes at distance $δ-1$ will send their current counter value $c$ to their children. These children will accept these tokens carrying the counter value $c$ from their parent only if their own counter value $c_u$ is different. However, if a child does not accept the Token message, then $c_u = c$. If a child does accept the token, it will immediately set its counter value $c_u$ to be $c$. Therefore, since all nodes at distance $δ$ from the source must be children of nodes at distance $δ-1$ by definition, all nodes at distance $δ$ will receive and accept the counter value $c$. Since the counter value is sent and received in constant time, all nodes will have received the new counter value in time proportional to the height of the tree, $O(h)$. 
Lemma 2 Let $G = (V, E)$ be a flow network with source $s$ and sink $t$, and $f$ be a preflow in $G$. Then, for any overflowing vertex $u$, there is a simple path from $u$ to $s$ in the residual network $G_f$.

Proof: Let $U$ be the set of nodes $v$ that there exists a simple path from $u$ to $v$ in $G_f$, and let $\bar{U} = V - U$.

Assume that there is no simple path from vertex $u$ to source node $s$, so $s \notin U$.

For each pair of vertices $v \in U$ and $w \in \bar{U}$, we claim that $f(w, v) \leq 0$ (proof: if $f(w, v) > 0$, then $f(v, w) < 0$, which implies that $c(v, w) = c(v, w) - f(v, w) > 0$. Hence, there exists an edge $(v, w) \in E_f$, and therefore there is a simple path from $u$ to $w$ in $G_f$, which contradicts our choice of $w$. So $f(w, v) \leq 0$).

Thus, we must have $f(\bar{U}, U) \leq 0$, since every term in this implicit summation is nonpositive. So $e(U) = f(V, U) = f(\bar{U}, U) + f(U, U) = f(\bar{U}, U) \leq 0$. Because excesses are nonnegative for all vertices in $V - \{s\}$; Because we assumed that $s$ is not a member of $U$, so $U \subseteq V - \{s\}$, we have $e(v) = 0$ for every node $v \in U$. It means that $e(u) = 0$, which contradicts the fact that $u$ is an overflowing vertex. So, there exists a simple path from $u$ to source node $s$ in the residual network $G_f$.

Lemma 3 During the execution of preflow_push algorithm on any flow network $G = (V, E)$, the number of basic operations is $O(|V|^2|E|)$.

Proof: In the preflow_push algorithm, there are two basic operations: Lift and Push. and there are two kinds of Push operations: 1. saturating push, if edge $(u, v)$ becomes saturated (i.e., $c(u, v) = 0$) after push operation. 2. nonsaturating push (i.e., $c(u, v) > 0$ after push operation). To prove lemma 3, we have to prove following:
1) bound on lift operation:

**Proof:** According to the height function, the height of the source node \( s \) and the height of the sink node \( t \) never change through the execution of preflow_push algorithm. \( h(s) = |v| \), and \( h(t) = 0 \).

Since a node is lifted only when it is overflowing, for every overflowing vertex \( u \in V - \{s, t\} \), we know by lemma 2 that there exists a simple path \( p = (v_0, v_1, \ldots, v_k) \) from node \( u \) to source node \( s \) in \( G_f \), where \( v_0 = u \), \( v_k = s \) and \( k \leq |V| - 1 \) since \( p \) is a simple path. For \( I = 0, 1, \ldots, k-1 \), \( (v_I, v_{I+1}) \in E_f \), according to the height function that \( h[v_I] \leq h[v_{I+1}] + 1 \), expanding these inequalities over path \( p \) yields \( h[u] = h[v_0] \leq h[v_k] + k \leq h[s] + (|V| - 1) = 2|V| - 1 \).

In the flow network, a vertex \( u \in V - \{s, t\} \) may be lifted by operation \( \text{lift}(u) \). Operation \( \text{lift}(u) \) increases \( h[u] \). Since the height of node \( u \) is initially 0 and grows to at most \( 2|V| - 1 \) (proved above). Each node \( u \) is lifted at most \( 2|V| - 1 \) times. Since there is a total of \( |V| - 2 \) nodes that may be lifted, the total lift operations performed is at most \( (2|V| - 1) (|V| - 2) < 2|V|^2 \).

2) Bound on saturation pushes

**Proof:** For any pair of vertices \( u, v \in V \), consider saturating pushes from \( u \) to \( v \) and from \( v \) to \( u \). If there is any such pushes, at least one of \( (u, v) \) and \( (v, u) \) is actually an edge in \( E_f \). Suppose that a saturating push from \( u \) to \( v \) has occurred, now \( c(u,v) = 0 \). In order to have another saturating push from \( u \) to \( v \), there must have a push from \( v \) to \( u \) to make \( c(u,v) > 0 \), which cannot happen until \( h[v] \) is increased by at least 2. Likewise, \( h[u] \) must increase by at least 2 for the next saturating push from \( u \) to \( v \).
Consider the sequence $A$ of integers given by $h[u] + h[v]$ for each saturating push that occurs between $u$ and $v$. When first saturating push in either direction between node $u$ and $v$ occurs, $h[u] + h[v] \geq 1$ must hold. When the last such push occurs, $h[u] + h[v] \leq (2|V|-1) + (2|V| -2) = 4|V| -3$ must hold. So the first and last possible integer in $A$ is at least 1 and at most $4|V| -3$, respectively. Furthermore, by the argument from the previous paragraph, at most every other integer can occur in $A$. Thus, the total number of integers in $A$ is at most $((4|V| -3)/2 + 1 = 2|V| - 1$. The total number of the saturating pushes between vertices $u$ and $v$ is therefore at most $2|V| -1$. So the total number of the saturating pushes in flow network $G(V, E)$ is $(2|V| -1)|E| < 2|V||E|.$

3) Bound on nonsaturating pushes

**Proof:** Define a potential function $\Phi = \sum_{v \in X} h[v]$, where $X \subseteq V$ is the set of overflowing vertices. Initially, $\Phi = 0$. Since the maximum possible height of a vertex is $2|V| -1$, so a lift operation on a vertex $u$ increases $\Phi$ by at most $2|V| -1$. For a saturating push from $u$ to $v$, since no height is changed, only vertex $v$ can possibly become a new overflowing vertex, and $h[v] \leq 2|V| -1$, so a saturating push increases $\Phi$ by at most $2|V| -1$. For nonsaturating push from $u$ to $v$, after push $u$ is no longer overflowing, $v$ is overflowing, and $h[u] = h[v] + 1$, so a nonsaturating push decreases $\Phi$ by at least 1.

During the execution of the algorithm, $\Phi$ can be increased at most $((2|V|-1)^2 + (2|V|-1)(2|V||E|) \leq 4|V|^2(|V| + |E|)$ by lift operations and saturating pushes, and can only be decreased by nonsaturating pushes. Since initially $\Phi = 0$, and $\Phi \geq 0$ throughout the execution of the algorithm, the total number of the nonsaturating pushes is at most $4|V|^2(|V| + |E|)$. 
From above three arguments, we conclude that the number of basic operations is $O(|V|^2|E|)$.

**Lemma 4** The implementation of preflow_push algorithm runs in $O(|V|^2|E|)$ time on any flow network $G(V, E)$.

**Proof:** From the code of this generic preflow_push algorithm, we can see that every lift takes $O(|V|)$ time and every push takes $O(1)$ time. According to the bound of the lift and push operations by Lemma 3, the implementation of preflow_push algorithm runs in $O(|V|^2|E|)$ time on any flow network $G(V, E)$.

**Lemma 5** A fresh counter will be produced in $O(|V|^2|E|)$ time if a system starts at arbitrary state of reset phase.

**Proof:** If the system starts at a state of reset phase, source node $s$ will change $turn_s$ from value $reset$ to value $preflow-push$ only when $FINISHED(s)$ is true. This yields two cases:

- **case 1:** The source node $s$ is initialized with $FINISHED(s)$ being true. In this case, $turn_s$ is assigned to $preflow-push$ in $O(1)$ time.

- **case 2:** The source node $s$ is expecting at least one message from a child and thus $FINISHED(s)$ is false at $s$. Thus, the source node $s$ will continually send $Tokens$ with current counter value $c_s$ to its children. By lemma 1, this $c_s$ will have reached all nodes in the network within $O(h)$ time. Once a leaf node receives the value $c_s$, it will begin sending $Tokens$ to its parent with the same counter value. All parents will accept these values since they will also hold the counter value $c_s$ by Lemma 1. In the same way all of these nodes will send same $Tokens$ to their parents and by induction on the maximum distance
of a node from a leaf, all nodes up to the sources node \( s \) will receive these tokens with the value \( c_s \). Once the source node \( s \) has received tokens with value \( c_s \) from all of its children, \textbf{FINISHED}(s) is true. Clearly, the tokens travel up the tree in the same time as they travel down the tree. \( O(h) \). Therefore, the turn, is set to \textit{preflow_push} in \( O(h) \) time from any arbitrary state in reset phase.

When \textit{turn} is set to \textit{preflow_push}, the system enters preflow_push phase. Every node in the network has either the same counter value or different counter value, and has either a correct initiation or an incorrect initiation. So there are three cases:

\textbf{case 1:} Every node has the same counter value and is correctly initiated. In this case, all nodes are initialized correctly for preflow_push protocol. By Lemma 4, the implementation of preflow_push algorithm will terminate in \( O(|V|^2 |E|) \) time. So a fresh counter will be produced in \( O(|V|^2 |E|) \) time by using \textbf{CHOOSE} (MAX, \( c_s \)) function.

\textbf{case 2:} Every node has the same counter value, but some nodes are not correctly initialized. This leads to two subcases: 1) incorrect initialization does not violate rule \( c_T (u, v) + c_T (v, u) = c(u, v) + c(v, u) \) and rule \( f(u, v) = - f(v, u) \). 2) incorrect initialization does violate at least one of the above two rules. In subcase 1, preflow_push algorithm will still terminate in \( O(|V|^2 |E|) \) time. So a fresh counter will be produced in \( O(|V|^2 |E|) \) time by using \textbf{CHOOSE} (MAX, \( c_s \)) function. In subcase 2, according to preflow_push algorithm, node \( v \) that violates the rule(s) will send a reset request message attached with its current counter value. This message will send up to the source node \( s \), since they all have the same counter value. The time from message sent by \( v \) to message received by source node \( s \) is \( O(h) \) by lemma 1. After receiving \textit{reset_request} message, the source
node s calls REQUEST_RESET(0) procedure and gets a fresh counter value with the
time complexity O(h).

case 3: some or all nodes have different counter value and the initialization may
or may not violate the rule(s). 1) If the initialization does not violate the previous
mentioned two rules, when the node v, the first node encountered on the way from s
down to a leaf which has different counter value from c_s, received
(turn,copy_flag,counter) message from its parent, the node will call procedure
T_RECEIVE_u(turn, copy_flag, counter). In this procedure, c_v will be set to the counter
received and a reset_request message will be sent to its parent. This message will reach
the source node s in O(h) time. 2) If there is such a node u that all parents and
grandparents and u itself have the same counter value as the source node does, but u
violates the rule(s), according to PREFLOW_PUSH algorithm, u will send a
reset_request message to its parent, and this message will be finally received by source
node s in O(h) time. In either above case, node s will produces a new counter in O(h)
time.

After the new counter is produced, the system will enter the reset phase again
and then enter the preflow_push phase in O(h) time. Since this new counter is probably
not a fresh counter, the system will fall into one of the three cases we just argued. If this
time it falls into the first two cases, a fresh counter will be produced in O(|V|^2 |E|) time.
If it falls into the third case again, the circle continues. Because the source node is the
only node that produces the new counter, and there is at most a total of |V| different
counter values in the network, in the worst situation the third case will be continued in
|V| - 1 times and the last time it will fall into one of the first two cases. Therefore, the
time complexity will be \((|V| - 1)*O(h) + O(|V|^2 |E|) = O(|V|^2 |E|)\).

So this proves that a fresh counter will be produced in \(O(|V|^2 |E|)\) time if a system
starts at any arbitrary state of the reset phase.

**Corollary 1:** A fresh counter can be produced in \(O(|V|^2 |E|)\) time if a system starts at any
arbitrary state of the preflow_push phase.

**Proof:** the proof is similar to that for Lemma 5, the only different is that it starts at the
preflow_push phase.

**Corollary 2:** A fresh counter can be produced in \(O(|V|^2 |E|)\) time from any arbitrary
state.

**Proof:** A network can start at any arbitrary state of either reset phase or preflow_push
phase. By Lemma 5 and Colollary 1, a fresh counter can be produced in \(O(|V|^2 |E|)\) time.
The corollary then follows.

**Lemma 6:** After a fresh counter is produced, a network will finish a correct reset phase
in \(O(h)\) time.

**Proof:** At the moment when a fresh counter is produced by the source node s, it can not
be held by any other node of the network. By Lemma 1, this fresh counter will reach all
nodes in the network in \(O(h)\) time, and during the period of time when a node receives
this fresh counter for first time, it will set its counter value to this fresh counter value and
correctly reset its local variables in \(O(1)\) time. When a leaf node receives this fresh
counter, sets its counter value to this fresh counter, and correctly resets its local variables
, it will begin sending the (turn, copyflag, counter) message to its parent. All leaf’s
parents will accept these same messages since they all have the same counter value. So in another \(O(h)\) time, the whole network will finish a correct reset, and the source node \(s\) will detect that the reset phase is finished. The total time complexity for completing this correct reset after a fresh counter is produced is \(O(h)\).

**Lemma 7:** After a correct reset phase, a network will enter the preflow\_push phase and the phase will terminate with a correct maximum flow in \(O(|V|^2|E|)\) time.

**Proof:** When a source node \(s\) detects that the reset phase is finished, the node immediately changes its \(\text{turn}\) variable to \(\text{preflow\_push}\). The entire network starts its preflow\_push phase. By Lemma 4, preflow\_push algorithm will terminate in \(O(|V|^2|E|)\) time on any flow network.

**Lemma 8:** Once a correct maximum flow is calculated in the network, regardless of any perturbations to the network, the correct flow will be copied into the \(f_{\text{final}}\) variables and remains there until the next update.

**Proof:** Once the correct maximum flow is calculated, the preflow\_push phase terminates and the reset phase starts. During the reset process, \(f(u, v)\) variables map to the \(f_{\text{final}}(u, v)\) at each node \(u\). This only happens in the \(\text{LOCAL\_RESET}\) function during the reset phase. By Lemma 7, by the time that next reset begins it will give out another correct maximum flow. When such a cycle goes on, the correct maximum flow will always be copied into the \(f_{\text{final}}(u, v)\) variables.

**Theorem 1:** The protocol given above is a correct maximum flow protocol on any flow network with \(O(|V|^2|E|)\) stabilization time.
**Proof:** Source node $s$ gets a fresh counter value in $O(|V|^2 |E|)$ time no matter what state it starts at by Corollary 2. Once a fresh counter value is produced, a correct reset will be achieved in $O(h)$ time by Lemma 6. After a correct reset, the correct maximum flow will be calculated in $O(|V|^2 |E|)$ time. Finally, during the next reset $f_{\text{final}}(u, v)$ variables take the correct net flow and hold their values according to Lemma 8. This procedure takes $O(h)$ time. Hence, the algorithm is correct and it stabilizes in $O(|V|^2 |E|) + O(h) + O(|V|^2 |E|) + O(h) = O(|V|^2 |E|)$ time.
Chapter 3

CONCLUSION

In this thesis, we presented a self-stabilizing algorithm for maximum flow problem. We have proved that the algorithm correctly computes the maximum flow in the network in a finite number of steps regardless of the network's initial state, and we also have proved that the algorithm can recover from transient faults in a finite number of steps.

There exists another self-stabilizing algorithm for the maximum flow problem. The algorithm was given by Sukumar Ghosh, Arobinda Gupta and Sriram Pemmaraju [17]. The algorithm is based on finding the shortest augment path method. However, our algorithm has following five advantage over their algorithm:

1) Their algorithm only works for the network which is loop-free (i.e., if edge \((i,j) \in E\), then edge \((j,i) \notin E\)). Our algorithm works for every general network.

2) Their algorithm needs some shared variable between neighbor processes. Our algorithm does not.

3) Their algorithm assumes that all actions are atomic. Our algorithm only assumes that every statement is atomic.

4) Their algorithm assumes that existence of a central scheduler that schedules the enabled guards in the system; our algorithm does not need a central scheduler.
5) They have not completely analysed the time complexity of their algorithm, but they do know that the rule 1 of their algorithm can be executed \(O(2^{|V|})\) times in the worst case. The time complexity of our algorithm is \(O(|V|^2 |E|)\).

As mentioned previously, our algorithm is based on generic preflow-push method. In this method the active nodes' push/lift operations can be executed in any order. We can improve the time complexity of the algorithm by specifying different rules for selecting active nodes for the push/lift operations. The bottleneck operation in the generic preflow-push algorithm is the number of nonsaturating pushes. Several specific rules for examining active nodes, such as FIFO, highest height first, and excess scaling, can produce substantial reductions in the number of nonsaturating pushes. So, we can use those schemes to improve our self-stabilizing maximum flow algorithm in the future.
process s

const n : { n is the number of processes in the network }

var turn : { reset, preflow_push },
c : 0 .. MAX

par j : 1 .. n-2

begin

\[ \text{turn}_s = \text{"preflow\_push"} \wedge \rightarrow \text{PREFLOW\_FINISHED} \]
\[ \rightarrow \text{PRE\_FLOW\_PUSH}(s) \]

\[ \square \text{turn}_s = \text{"preflow\_push"} \wedge \text{PREFLOW\_FINISHED} \]
\[ \rightarrow \text{REQUEST\_RESET}(\text{copy\_flags},) \]

\[ \square \text{turn}_s = \text{"reset"} \wedge \text{FINISHED}(s) \rightarrow \text{turn}_s := \text{"preflow\_push"} \]

\[ \forall v \{ s = \text{parent}_v \rightarrow \text{SEND}_s,(\text{turn}_s,\text{copy\_flags},c_v) \} \]

\[ \square \text{recv} < \text{message of (turn, copy\_flag, counter)} > \text{from < child process } v > \]
\[ \rightarrow \text{T\_RECEIVE}_s,(\text{turn, copy\_flag, counter}) \]

\[ \square \text{recv} < \text{message of (flow, counter)} > \text{from < child process } v > \wedge \]
\[ \text{turn}_s = \text{"preflow\_push"} \rightarrow \text{F\_RECEIVE}_s,(\text{flow, counter}) \]

\[ \square \text{recv} < \text{message of (reset\_request, counter)} > \text{from < child process } v > \wedge \]
\[ c_s = \text{counter} \rightarrow \text{REQUEST\_RESET}(0) \]

end

Figure 1: Protocol for source node s
process $p[u : 1..n-2]$ /* for every node in the network, except node $s$ and node $t$ */

```
var turn$_u$: {reset, preflow_push};
c$_u$: 0 .. MAX
begin
  turn$_u$ = "preflow_push" $\rightarrow$ \textbf{PRE\_FLOW\_PUSH}($u$)
  $\square$ u = parent$_v$ $\lor$ ($v = \text{parent}_u \land$ turn$_u$ = "reset" $\land$ \textbf{FINISHED}($u$))
  $\rightarrow$ \textbf{SEND}$_{u,v}$(turn$_u$,copy\_flag$_u$,c$_u$)
  $\square$ recv $<$ message of (turn, copy\_flag, counter) $>$ from $<$ process $v$ $>$
  $\rightarrow$ \textbf{T\_RECEIVE}$_{u,v}$(turn, copy\_flag, counter)
  $\square$ recv $<$ message of (flow$_t$, counter) $>$ from $<$ child process $v$ $>$
  $\rightarrow$ \textbf{F\_RECEIVE}$_{u,v}$(flow$_t$, counter)
  $\square$ recv $<$ message of (reset\_request, counter) $>$ from $<$ child process $v$ $>$ $\land$
  c$_u$ = counter $\rightarrow$ \textbf{SEND}$_{u,\text{parent}_u}$(reset\_request, counter)
end
```

**Figure 2:** Protocol for processes other than $s$ and $t$
process t

    var turn : { reset, preflow_push };  c : 0 .. MAX

begin
    turn_t = "preflow_push" → PRE_FLOW_PUSH(t);
    flow := { ∑ f(t, v) | v ∈ N_t }; SEND_{t, parent}(flow, c_t)
    □ t = parent, v ( v = parent, ∧ turn_t = "reset" ∧ FINISHED(t))
        → SEND_{t, v}(turn, copy_flag_t, c_t)
    □ recv < message of (turn, copy_flag, counter) > from < process v >
        → T_RECEIVE_{t, v}(turn, copy_flag, counter)
    □ recv < message of (reset_request, counter) > from < child process v > ∧
        c_t = counter → SEND_{t, parent}(reset_request, counter)

end

Figure 3: Protocol for sink process t
The state of each node $u$ consists of:

- $N_u$ the set of neighbors of the node $u$ in the network.
- parent$_u$ the id of the parent node of node $u$.
- $c_u$ a counter of node $u$.
- reset_finish_token_expected$_u(v)$ a Boolean flag for each child $v$ of $u$; True indicates $u$ is expecting a token from $v$.
- $e_u$ the excess flow stored at node $u$.
- $h_u$ the height of node $u$.
- $c(u,v)$ non-negative capacity on edge $(u,v)$.
- $c(u,v)$ residual capacity on edge $(u,v)$.
- $d(u,v)$ temporary containing the units of flow to be pushed.
- $f(u,v)$ netflow from node $u$ to node $v$.
- $f_{final}(u,v)$ netflow from $u$ to $v$, which is gotten from last complete computation of maximum flow.
- flow store latest maximum flow information sent by node $t$. This variable is only used in node $s$.
- turn$_u$ turn can take one of two strings: reset or preflow_push.
- copy_flag$_u$ to control netflow to copied to $f_{final}(u,v)$ or not.
- excess_flow$(u)$ true when $e_u = \{ \sum f(u,v) \mid v \in N_u \}$, otherwise it is false.
- rel_f_c_c$(u,v)$ true when $f(u,v) = c(u,v) - c(u,v)$, otherwise it is false.
- skew_sym$(u,v)$ true when $f(u,v) = -f(v,u)$, otherwise it is false.
- u_pushed$(u,v)$ true when $f(u,v) + f(v,u) = d(u,v)$, otherwise it is false.
- v_pushed$(u,v)$ true when $f(u,v) + f(v,u) = d(v,u)$, otherwise it is false.
- u_and_v_pushed$(u,v)$ true when $f(u,v) + f(v,u) = d(u,v) + d(v,u)$, otherwise it is false.
- rel_c_c$(u,v)$ true when $c(u,v) + c(v,u) = c(u,v) + c(v,u)$, i.e., true when
  \[
  \text{rel_f_c_c}(u,v) \wedge \text{rel_f_c_c}(v,u) \wedge \neg \text{u_pushed}(u,v) \wedge \neg \text{v_pushed}(u,v) \wedge \\
  \neg \text{u_and_v_pushed}(u,v) \wedge \neg \text{skew_sym}(u,v),
  \] otherwise it is false.

Figure 4: The variables used in each node
FINISHED(u) {\*Boolean function; return true when not expecting tokens from any children \*}

\begin{verbatim}
begin
  if every reset_finish_token_expected,(k) = false \quad return true
  else \quad return false
\end{verbatim}

PRE_FLOW_FINISHED {\* Boolean function used to check whether preflow_push phase is finished or not. \*}

\begin{verbatim}
begin
  copy_flags := 1;
  if flow = \{ \sum f(s,v) | v \in N_s \} \quad return true
  if flow > \{ \sum f(s,v) | v \in N_s \} \quad do copy-flags := 0; \quad return true
  else \quad return false
\end{verbatim}

REQUEST_RESET(copy_flag) {\* For source node s only \*}

\begin{verbatim}
begin
  copy_flags := copy_flag;
  LOCAL_RESET_s(copy_flag_s); \quad {\* reset node s \*}
  turn_s := "reset"; \quad {\* to indicate network is in reset phase. \*}
  c_s := \text{CHOOSE}(\text{MAX}, c_s); \quad {\* choose a fresh counter number. \*}
  for every child k of s \quad do \quad reset_finish_token_expected_s(k) := true;
  flow := 0
\end{verbatim}

T_RECEIVE_s,(turn, copy_flag, counter) {\* for node s only. Action after receiving turn message from child node v \*}

\begin{verbatim}
begin
  if (c_v = counter \land turn_v = "reset" \land turn = "reset")
    reset_finish_token_expected_s(v) := false
\end{verbatim}

T_RECEIVE_u,(turn, copy_flag, counter) {\* For every node other than node s \*}

\begin{verbatim}
begin
  if v = parent_s \quad {\* node u received turn message from its parent \*}
    if c_u \neq counter \land turn = "reset" \quad {
      turn_u := turn;
      copy_flag_u := copy_flag;
      c_u := counter;
      LOCAL_RESET_u(copy_flag);
    }
\end{verbatim}
for every child $k$ of $u$
    \[ \text{reset\_finish\_token\_expected}_u(k) := \text{true} \]
if $c_u = \text{counter} \land \text{turn} = \text{"preflow-push"}$
    \[ \text{turn}_u := \text{"preflow-push"} \]
if $c_u \neq \text{counter} \land \text{turn} = \text{"preflow-push"}$
    \[
    \begin{align*}
    c_u & := \text{counter}; \\
    \text{send}_u\_\text{parent}(\text{reset-request}, c_u) & \\
    \end{align*}
    \]
elseif $u = \text{parent}_v \land c_u = \text{counter} \land \text{turn} = \text{"reset"} \land \text{turn}_u = \text{"reset"}$
    \[
    \begin{align*}
    \text{reset\_finish\_token\_expected}_u(v) & := \text{false} \\
    \end{align*}
    \]
\[ \text{end} \]

**F\_RECEIVE}_{s,v}(flow_t, \text{counter})** (*action for node $s$ received flow message from its child node $v$*)
\[
\begin{align*}
\text{begin} & \\
\text{if} & \quad c_s = \text{counter} \quad \text{then} \quad \text{flow} := \text{flow}_t \\
\text{end} & \\
\end{align*}
\]

**F\_RECEIVE}_{u,v}(flow_t, \text{counter})** (*for every other node*)
\[
\begin{align*}
\text{begin} & \\
\text{if} & \quad c_u = \text{counter} \quad \text{then} \quad \text{SEND}_u\_\text{parent}(\text{flow}_t, \text{counter}) \\
\text{end} & \\
\end{align*}
\]

**LOCAL\_RESET}_{s}(\text{copy\_flag})** (*for node $s$ only*)
\[
\begin{align*}
\text{begin} & \\
\text{if} \quad \text{copy\_flag} = 1 \quad \text{then} \quad \text{COPT\_NETFLOW}(s) \\
\quad h_s &= |V|; \quad \{ \text{*set the height of $s$ to the number of nodes in the network.*} \} \\
\quad e_s &= 0; \\
\quad \text{for every } v \in N_s \quad \text{do} \quad \{ \quad f(s, v) = c(s, v); \quad c_r(s, v) = 0 \quad \} \\
\text{end} & \\
\end{align*}
\]

**LOCAL\_RESET}_{t}(\text{copy\_flag})** (*for sink node $t$ only*)
\[
\begin{align*}
\text{begin} & \\
\text{if} \quad \text{copy\_flag} = 1 \quad \text{then} \quad \text{COPT\_NETFLOW}(t) \\
\quad h_t := 0; \quad e_t := 0; \\
\text{for every } v \in N_t \quad \text{do} \quad \{ \quad f(t, v) := 0; \quad c_b(t, v) := c(t, v) \quad \} \\
\text{if} \quad t \in N_e \quad \text{then} \quad \{ \quad f(t, s) := - c(s, t); \quad c_b(t, s) := c(t, s) - f(t, s) \quad \} \\
\text{end} & \\
\end{align*}
\]

**LOCAL\_RESET}_{u}(\text{copy\_flag})** (*for other nodes*)
\[
\begin{align*}
\text{begin} & \\
\text{if} \quad \text{copy\_flag} = 1 \quad \text{then} \quad \text{COPT\_NETFLOW}(u) \\
\quad h_u := 0; \\
\end{align*}
\]
eu := 0;
for every v ∈ Nu do { f(u, v) := 0; c(u, v) := c(u, v) }
if u ∈ Ns then { eu := c(s, u); f(u, s) := -c(s, u);
c(u, s) := c(u, s) - f(u, s) }
end

COPY_NETFLOW(u) {* for every node in the network *}
begin
for every v ∈ Nu do ffina i(u, v) := f(u, v)
end

PRE_FLOW_PUSH(u)
begin
excess_flow(u) ∧ ∀v(skew_sym(u, v)) ∧ ∃v(rel_f_c_c(u, v) ∧
rel_f_c_c(u, v) ∧ eu > 0 ∧ c(u, v) > 0 ∧ hu = hv + 1) → PUSH(u, v)
□ ∃v((v_pushed(u, v) ∨ u and_v_pushed(u, v)) ∧ rel_f_c_c(u, v) ∧
rel_f_c_c(v, u) → COLLECT_FLOW(v, u)
□ eu > 0 ∧ ∀v(c(u, v) > 0 ∧ hu ≤ hv) ∧ excess_flow(u) ∧
∀v(skew_sym(u, v)) ∧ ∃v(rel_f_c_c(u, v)) → hu := 1 + min{hv | c(u, v) > 0}
□ ~excess_flow(u) → eu := Σf(v, u)
□ ~rel_f_c_c(u, v) → f(u, v) := c(u, v) - c(u, v)
□ ~rel_f_c_c(u, v) → SEND_u_paren(reset_request, c_u)
□ ∃v( eu > 0 ∧ c(u, v) > 0 ∧ hu > hv + 1) → hu := 1 + min { hv | c(u, v) > 0 }
end

PREFLOW_PUSH(s)
begin
exists v(pushed(s, v) ∧ rel_f_c_c(s, v) ∧ rel_f_c_c(v, s)) → COLLECT_FLOW(v, s)
□ eu ≠ 0 → eu := 0
□ hu ≠ |V| → hu := |V|
□ ~rel_c_c(s, v) → REQUEST_RESET(0)
□ ~rel_f_c_c(s, v) → f(s, v) = c(s, v) - c(s, v)
end

PREFLOW_PUSH(t)
begin
exists v(pushed(t, v) ∧ rel_f_c_c(u, v) ∧ rel_f_c_c(v, u)) → COLLECT_FLOW(v, u)
□ eu ≠ 0 → eu := 0
\( h_t \neq 0 \rightarrow h_t := 0 \)
\( \neg \text{rel}_c \circ (u,v) \rightarrow \text{SEND}_{i, \text{parent}}(\text{reset \_request}, c_i) \)
\( \neg \text{rel}_f \circ c(t,v) \rightarrow f(t,v) = c(t,v) - c_i(t,v) \)

PUSH\((u,v)\)
begin
\[ d(u,v) := \min(e_u, c(u,v)); \]
\[ e_u := e_u - d(u,v); \]
\[ f(u,v) := f(u,v) + d(u,v); \]
\[ c(u,v) := c(u,v) - f(u,v); \]
end

COLLECT\_FLO\(W(v,u)\)
begin
\[ e_u := e_u + d(v,u); \]
\[ f(u,v) := f(u,v) - d(v,u); \]
\[ c(u,v) := c(u,v) + f(u,v); \]
end

**Figure 5**: All support functions and preflow\_push code
Bibliography


