Perfect Token Distribution on Trees*

Luciano Margara1, Alessandro Pistocchi2, and Marco Vassura3

1 Computer Science Department, University of Bologna. margara@cs.unibo.it
2 Computer Science Department, University of Bologna. pistocchi@cs.unibo.it
3 Computer Science Department, University of Bologna. vassura@cs.unibo.it

Abstract. Load balancing on a multi-processor system consists of redistributing tasks among processors so that all processors end up with roughly the same amount of work to perform. The token distribution problem is a variant of the load balancing problem where each task has unit-size and it represents an atomic element of work. We present an algorithm for computing a perfect token distribution (each processor has either \([T/N]\) or \([T/N]\) tasks, where \(N\) is the number of processors and \(T\) is the number of tasks scattered among processors) on distributed tree-connected networks having worst-case running time \(O(TD)\) (\(D\) denotes the diameter of the tree). The number of token exchanges exceeds the optimum by at most \(O(D\min\{T, N\})\).

In order to compute a perfect token distribution each node \(v\) must be able to store \(\Theta(d_v(\log T + \log N))\) bits, where \(d_v\) is the degree (number of adjacent nodes) of \(v\). This is the first fully decentralized algorithm for computing perfect token distributions on arbitrary tree-connected networks which does not receive as input any kind of aggregate information about the network (e.g., number of nodes or total number of tokens).

1 Introduction.

The performance of a distributed network crucially depends on dividing up work effectively among its processing elements [8]. This type of load balancing problem has been studied in many different models. The basic idea in all of the models is to evenly redistribute initial job load among processors (static balancing) and to keep load distribution as balanced as possible during time (dynamic balancing). Many variants of the load balancing problem have been proposed and widely investigated in the literature [1,2,3,4,6,9,10,11].

In this paper we consider a basic variant of the static load balancing problem, called token distribution problem, restricted to tree-connected networks. A tree-connected network is represented by a tree with \(N\) nodes. Each node represents

* This work was partially supported by the Future & Emerging Technologies unit of the European Commission through Project BISON (IST-2001-38923).
a processing element of the network and possesses a number of jobs of unit-size (tokens) to be processed. The total number of tokens is denoted by $T$. In a single message a token can be moved from any node to any other adjacent node in the tree. No token is created or destroyed during the redistribution process. The goal is to redistribute tokens across the tree so that each node ends up (being aware of that) with either $[T/N]$ or $[T/N]$ tokens (perfect token distribution). We adopt asynchronous single-port communication model with uni-directional communication links. Nodes are anonymous, i.e. they do not have identification labels. Synchronization between nodes (executing the same code in parallel) is achieved by exchanging messages.

**Previous results.** For general networks Ghosh et al. [4] analyze two algorithms which reduce the maximum difference in tokens between any two nodes (called "discrepancy") to at most $O(d^2 \log N)/\alpha$, where $d$ is the maximum degree of the nodes of the network, $N$ is the number of nodes in the network, and $\alpha$ is the edge expansion of the network. Many results have been carried out on specific network topologies. For ring networks Gehrke, Plaxton, and Rajaraman [3] give an algorithm having an asymptotically optimal message complexity which converges to a perfectly balanced state.

For meshes and torus Houle et al. [7] give an algorithm that reduces the discrepancy to the minimum degree of the nodes of the network and that runs in worst-case optimal time. The same algorithm used for complete binary trees obtains in the worst case a discrepancy equal to the height of the tree [6].

For arbitrary trees Houle, Symvonis, and Wood [5] give an algorithm for computing a perfect token distribution assuming that each node at the beginning of the computation knows the number of nodes in the tree.

**Our results.** We present a fully decentralized algorithm for tree-connected networks which computes a perfect token distribution in time $O(TD)$ without receiving as input any additional (aggregate) information about the network such as the total number of nodes or the total number of tokens. The number of token exchanges made by our algorithm exceeds the optimum by at most $O(D \min(T,N))$. In addition, at the end of the redistribution process, all the nodes reach a distinguished final state and they are ready to start a new activity.

Our technique is based on a three phase approach.

- **Phase 1.** We compute (in a completely distributed way) the number of nodes of the tree and the number of tokens scattered across the tree in $\Theta(Dd)$, where $d$ is the maximum degree and $D$ is the diameter of the tree\(^4\).

- **Phase 2.** Let $v$ be any node of the tree and $T_i$, $1 \leq i \leq k$, be the $k$ sub-trees rooted at nodes adjacent to $v$. After Phase 1 $v$ knows the number of nodes and the number of tokens contained in each $T_i$. Taking advantage of this information each node having more than $T_i/N$ tokens is able to decide where to send its

\(^4\) in multi-port models such information can be computed in $\Theta(D)$. 
extra tokens. Nodes that have less than \([T/N]\) tokens wait until they receive enough tokens from their neighbors. After Phase 2 we get an "almost perfect" token distribution in which at least \(N - 1\) nodes have exactly \([T/N]\) tokens. Note that almost perfect token distributions have discrepancy at most \((T \mod N)\).

- **Phase 3.** We refine the distribution obtained after Phase 2 in order to make it perfect. To this extent, the node containing more than \([T/N]\) tokens (at most one node has this property) sends them to its neighbors. This procedure is then executed recursively by all the neighbors of \(v\).

The rest of the paper is organized as follows. In Sect. 2 we define some communication primitives that will be used in our algorithm. In Sect. 3 we present our algorithm while in Sect. 4 and 5 we sketch the proof of its correctness and we discuss its computational cost in terms of time, space, and number of token exchanges, respectively. Section 6 contains conclusions and a brief description of possible open problems.

## 2 Communication Primitives.

We define the following communication primitives:
- \(\text{reserve}(c)\) reserves channel \(c\) for communication and returns \(\text{busy}\) if \(c\) is already reserved;
- \(\text{release}(c)\) releases channel \(c\) reserved for communication;
- \(\text{send}(c, m)\) sends message \(m\) through the already reserved channel \(c\);
- \(\text{receive}(c)\) receives a message from channel \(c\) if a message was sent on channel \(c\) (channel needs not to be reserved). If no message is arriving from \(c\) the message returned by \(\text{receive}\) is \(\text{NULL}\). If two or more messages are sent on a channel then they are received one by one in the order they were sent.

Using these primitives we build functions:
- \(\text{waitAny}(c)\) interrogates every channel until a message is received, then returns the message received, saving the channel through which it arrived in the parameter \(c\);
- \(\text{wait}(c)\) interrogates channel \(c\) until a message is received, then returns the message received; \(\text{safeSend}(c, m)\) waits until \(c\) is not \(\text{busy}\), reserves it, sends \(m\) through \(c\), and then releases \(c\);
- \(\text{receiveToken}(c)\) receives a token from \(c\), if any. If a message is received updates local variables containing the number of tokens;
- \(\text{receiveTokensFromAll}()\) receives, using \(\text{receiveToken}(c)\), tokens sent from neighbors, if any;
- \(\text{sendToken}(c)\) sends, using \(\text{safeSend}(c, m)\) a token from the current node through \(c\) and updates the local variables containing the number of tokens.

## 3 The Algorithm.

Our algorithm consists of three phases: first it collects information about the tree structure and the initial token distribution, then uses this information for
obtaining an almost perfect token distribution, and finally it refines the solution by redistributing a small number of tokens.

In what follows we assume that each node $v$ can store $\Theta(d_v (\log T + \log N))$ bits of information.

3.1 Phase 1.

Each node $v$ has an internal state defined according to the values of two local variables, namely $S$ and $R$ which contain the number of messages sent and received by $v$ so far. Let $\deg$ be the degree $d_v$ of $v$. Both $S$ and $R$ are set to 0 at the beginning of the computation. According to the values of $S$ and $R$ we define 5 distinct states:

$S1.1: [S = 0 \text{ and } R < (\deg - 1)]$. All the nodes having degree greater than 1 start in this state. Nodes in this state are waiting for messages from their neighbors. Any node with $k$ neighbors waits for $k - 1$ messages and then changes state (from $S1.1$ to $S1.2$).

$S1.2: [S = 0 \text{ and } R = (\deg - 1)]$. Leaves start in this state. Each node with $k$ neighbors reaches this state after receiving $k - 1$ messages. A node in this state tries to send information about its subtree to the only neighbor from which it did not receive any message yet. If succeeding the node changes state (from $S1.2$ to $S1.3$). Otherwise, if the channel is busy then the last neighbor is sending subtree information. The node receives it and changes state (from $S1.2$ to $S1.4$).

$S1.3: [S = 1 \text{ and } R = (\deg - 1)]$. Nodes in this state are waiting for global information from the neighbor to which they sent their local information.

$S1.4: [S = 0 \text{ and } R = \deg]$. Only one node reaches this state: the one computing global information. In this state it sends such information to all its neighbors and then ends.

$S1.5: [S = 1 \text{ and } R = \deg]$. In this state a node has already received global information and forwards it to all its neighbors and then ends.

In phase 1 each message $msg$ consists of two integer numbers, referred to as $msg.T$ and $msg.N$ representing the number of tokens and of nodes of the entire subtree rooted at the sender of $msg$. We assume that channels at each node $v$ are numbered from 1 to $\deg$.

3.2 Phase 2.

In phase 2 we start moving tokens.

From now on we will refer to the node which received local information from all its neighbors as the root of the tree. Every node has two arrays, namely $\text{subTree}N$ and $\text{subTree}T$, containing the total number of nodes and the total number of tokens in the subtrees rooted at its neighbors. Each node, knowing total number of tokens $T$ and of nodes $N$ in the tree, computes the average number of tokens per node $\text{Avg} = \lfloor T / N \rfloor$. All nodes send or wait for tokens until they and the nodes in their subtrees contain exactly $\text{Avg}$ tokens, except for the
Phase 1

initialization
\[ \text{deg} \leftarrow d_v \] // neighbors of \( v \)
\[ T_v \leftarrow \text{tokens at} v \] // tokens in subtree rooted at \( v \)
\[ N_v \leftarrow 1 \] // nodes in subtree rooted at \( v \)
\[ R \leftarrow 0 \] // messages received by \( v \)
\[ S \leftarrow 0 \] // messages sent by \( v \)
\[ \text{parent} \leftarrow 0 \] // the parent of the node

subTreeT array of degree NULL elements // tokens in neighbor subtrees
subTreeN array of degree NULL elements // nodes in neighbor subtrees

while \( S < \text{deg} \) or \( R < \text{deg} \) do
  case \( S, \text{deg} \) of
    0, \text{deg} - 1:
      while \( R < (\text{deg} - 1) \):
        if \( \text{deg} \leftarrow \text{deg} - 1 \):
          \( \text{parent} \leftarrow \text{channel} j \) for which \( \text{subTreeT}[j] = \text{NULL} \) // the channel from which \( v \) did not receive msg

        if \( \text{msg} \leftarrow \text{wait}(\text{msg}) \) then // wait for a message from any neighbor
          \( T_v \leftarrow T_v + \text{msg} \).T // tokens in subtree rooted at \( v \)
          \( N_v \leftarrow N_v + \text{msg}.N \) // nodes in subtree rooted at \( v \)
          \( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
          \( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
          \( \text{parent} \leftarrow 0 \) // \( v \) is root, \( \text{parent} = 0 \)
          \( R \leftarrow R + 1 \) // \( v \) received a message
        else
          \( \text{release}(\text{parent}) \) // release channel parent
          \( S \leftarrow S + 1 \) // \( v \) sent a message
        endif
      endif
    else:
      \( \text{msg} \leftarrow \text{wait}(\text{msg}) \) // wait for the message
      \( T_v \leftarrow T_v + \text{msg} \).T // tokens in subtree rooted at \( v \)
      \( N_v \leftarrow N_v + \text{msg}.N \) // nodes in subtree rooted at \( v \)
      \( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
      \( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
      \( \text{parent} \leftarrow 0 \) // \( v \) is root, \( \text{parent} = 0 \)
      \( R \leftarrow R + 1 \) // \( v \) received a message
    endif
endwhile

S = 1, \text{deg}:
\( \text{msg} \leftarrow \text{wait}(\text{msg}) \) // wait for global message
\( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
\( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
\( T_v \leftarrow \text{msg}.T \) // total tokens of the tree
\( N_v \leftarrow \text{msg}.N \) // total nodes of the tree
\( R \leftarrow R + 1 \) // \( v \) received a message

S = 0, \text{deg}:
\( \text{msg} \leftarrow \text{wait}(\text{msg}) \) // wait for global message
\( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
\( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
\( \text{parent} \leftarrow 0 \) // \( v \) is root, \( \text{parent} = 0 \)
\( R \leftarrow R + 1 \) // \( v \) received a message

S = 1, \text{deg}:
\( \text{msg} \leftarrow \text{wait}(\text{msg}) \) // wait for global message
\( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
\( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
\( \text{parent} \leftarrow 0 \) // \( v \) is root, \( \text{parent} = 0 \)
\( R \leftarrow R + 1 \) // \( v \) received a message

S = \text{deg}:
\( \text{msg} \leftarrow \text{wait}(\text{msg}) \) // wait for global message
\( \text{subTreeT}[\text{parent}] \leftarrow \text{msg}.T \) // tokens in subtree rooted at \( v \)
\( \text{subTreeN}[\text{parent}] \leftarrow \text{msg}.N \) // nodes in subtree rooted at \( v \)
\( \text{parent} \leftarrow 0 \) // \( v \) is root, \( \text{parent} = 0 \)
\( R \leftarrow R + 1 \) // \( v \) received a message

endwhile
root of the tree which might contain extra tokens. Function \texttt{subTreeAvg(i)} is a simple function computing the average number of tokens in the subtree rooted at \(v_i\); it returns \(\text{subTree[i]}/\text{subTreeN[i]}\). Each node \(v\) in Phase 2 has an internal state represented by

- a boolean variable \texttt{subTreeBalanced} (true if all the subtrees rooted at children of \(v\) are balanced, false otherwise);
- an integer variable \(T_v\) storing the number of tokens at \(v\);
- an integer variable \texttt{parent} which is set to 0 if \(v\) is the root.

\textbf{S2.1:} \(\texttt{subTreeBalanced} = \text{false} \text{ and } T_v \leq \text{Avg}\). Nodes remain in this state receiving tokens and updating local variables until they have more than \text{Avg} tokens and then change state. (from S2.1 to S2.3).

\textbf{S2.2:} \(\texttt{subTreeBalanced} = \text{true} \text{ and } T_v < \text{Avg}, \text{parent} > 0\). Nodes in this state need not to exchange tokens with children, since all such subtrees have \text{Avg} number of tokens. They wait for tokens from their parent (a node in this state is not the root).

\textbf{S2.3:} \(\texttt{subTreeBalanced} = \text{false} \text{ and } T_v > \text{Avg}\). Nodes in this state need to balance their descending subtrees: they receive tokens from any node and then send tokens to subtrees with less than \text{Avg} number of tokens.

\textbf{S2.4:} \(\texttt{subTreeBalanced} = \text{true} \text{ and } T_v > \text{Avg}, \text{parent} > 0\). Nodes in this state have already balanced subtrees rooted at their children and send extra tokens towards their parents.

As we already mentioned, a node stops exchanging tokens if it has balanced its subtrees (\(\texttt{subTreeBalanced} = \text{true} \text{ and } T_v = \text{Avg}\)) (Fig. 1 part (b)) or if it has balanced the subtrees of all its children (\(\texttt{subTreeBalanced} = \text{true}\)) and it is the root (\(\text{parent} = 0\)) (Fig. 1 part (c)).

\subsection*{3.3 Phase 3.}

In this phase tokens keep on moving until a \textit{Finished} message is sent. In order to distinguish between tokens and finished messages we use two internal variables, namely \texttt{msg Token} and \texttt{msg Finished}. At any time if \texttt{msg Finished} = \text{true} then \texttt{msg Token} = \text{NULL}, else if \texttt{msg Finished} = \text{false} then \texttt{msg Token} contains a token.

In phase 3 extra tokens sent to the root during phase 2 are redistributed down the tree. The root (and all the other nodes recursively) sends its extra tokens to subtrees which can accept them.\(^5\) In this phase each node can be in one of the two following states:

\textbf{S3.1:} \(T_v > \text{Avg}+1 \text{ and } \text{parent} = 0\). Nodes in this state are root of a subtree and send their extra tokens to subtrees which can accept them.

\textbf{S3.2:} \(T_v \text{ any value} \text{ and } \text{parent} > 0\). Nodes in this state receive messages from their parent. If they receive tokens they update local variables. If they receive a \textit{Finished} message they set \textit{parent} to 0 and become a root.

When a node is root and has at most \text{Avg}+1 number of tokens it sends a \textit{Finished}

\(^5\) eventually, each node ends up with either \text{Avg} or \text{Avg}+1 tokens, then each subtree must contain less than \text{Avg}+1 tokens.
Phase 2
initialization
Avg ← |T_v/N_v|
T_v ← tokens at v
R ← 0
bigger ← \{\text{subTreeAvg}(i) > Avg, i \neq \text{parent}\}
smaller ← \{\text{subTreeAvg}(i) < Avg, i \neq \text{parent}\}
subTreeBalanced ← (bigger \neq \emptyset \text{ and smaller} \neq \emptyset)

while(snot(subTreeBalanced=true and T_v=Avg)
or (subTreeBalanced=true and parent=0)) do
  case (subTreeBalanced, T_v, parent) of
    subTreeBalanced = false, T_v < Avg, parent any value
      receiveTokensFromAll()
      // check for incoming tokens and update data
      bigger ← \{\text{subTreeAvg}(i) > Avg, i \neq \text{parent}\}
smaller ← \{\text{subTreeAvg}(i) < Avg, i \neq \text{parent}\}
      subTreeBalanced ← (bigger \neq \emptyset \text{ and smaller} \neq \emptyset)
      receiveToken(parent)
      // all sons have finished,
      // checking tokens coming from parent
    subTreeBalanced = false, T_v > Avg, parent any value
      receiveTokensFromAll()
      // check for incoming tokens and update data
      bigger ← \{\text{subTreeAvg}(i) > Avg, i \neq \text{parent}\}
      if(smallesr \neq \emptyset)
        j ← an element ∈ smaller
        sendToken[j]
        // send token to v_j and update data
      smaller ← \{\text{subTreeAvg}(i) < Avg, i \neq \text{parent}\}
      end
    subTreeBalanced ← (bigger \neq \emptyset \text{ and smaller} \neq \emptyset)
    subTreeBalanced ← (true, T_v < Avg, parent > 0)
    sendToken(parent)
    // send token to v_{parent} and update data
  endcase
endwhile

message to all its children. As proven in Sect. 4, eventually all nodes become a
root and receive a Finished message (Fig. 1 part (d)).

4 Proof of Correctness.

In this Section we prove the correctness of our algorithm.

4.1 Phase 1.

Let S_{1.final} be the final state of Phase 1 defined by |S = \text{deg} and R = \text{deg}|.

Theorem 1. At the end of phase 1, \forall v \in V we have that T_v = T \text{ and } N_v = N.
In addition, exactly one node (the root of the tree) has parent = 0 while each
other node has parent = i where i is the number of the channel which links the
node to its parent in the rooted tree.

Sketch of the proof: At the beginning of the computation, each node of degree
1 is in state S_{1.2} while all the other nodes are in state S_{1.1}. Then, each node
in state S_{1.2} sends its subtree information to nodes in state S_{1.1} and moves to
state S_{1.3}. Eventually each node in state S_{1.1} moves to state S_{1.2} and decides
which node is its parent. When state S_{1.2} contains only two nodes, one of them
moves to state S_{1.4}. All the other nodes go to state S_{1.3} and wait for global
information. The node in state S_{1.4} is the root, computes global information
Phase 3
initialization
p ← parent  // p is the parent of v

while (not Tv < Avg + 1 and parent = 0) do
  case (Tv, parent) of
  Tp > Avg + 1, parent = 0
    i ← a neighbor for which subtreeAvg[j] < Avg + 1, i ≠ p
    msg.token ← a token at v
    msg.finished ← false
    send(i, msg)  // send token to vi
    Tv ← Tv - 1  // tokens at v
    subtreeT[i] ← subtreeT[i] + 1  // tokens in subtree rooted at vi
  Tp any value, parent > 0
    msg ← receive(parent)
    if (msg.finished = true)  // if a message msg received
      parent ← 0  // the parent finished: become root
      end
      Tz ← Tz + 1
      subtreeT[parent] ← subtreeT[parent] - 1
    endif
  endcase
endwhile
for i ← 1 to deg do  // for all neighbors vi
  msg.token ← NULL
  msg.finished ← true
  if (i ≠ p)  // except the parent
    send(i, msg)  // send Finished to vi
  endif
endfor

and sends it to its neighbors. Neighbors receive such information, identify the
node which sent them as their parent, move to state S1.5, and then forward it to
their children. At the end of this process all the nodes receive global information
and set Tv = T and Nv = N. Eventually each node moves to state S1.final. □

4.2 Phase 2.

During Phase 1 we implicitly elect a leader, namely the only node with parent = 0.
We refer to this node as the root of the tree. Links to parent nodes are provided
by local variables parent.

Lemma 1. Let vj be a node different from the root of the tree and such that
subtreeAvg(j) ≥ Avg. Let v be any node of the subtree rooted at vj. Then at
the end of phase 2 v contains exactly Avg tokens.

Proof. We make the proof by induction on the height of the subtree:
Base case. If the subtree has height 0 (only one node) subtreeBalanced = true.
If the node has Avg tokens it will do nothing in Phase 2 and the lemma is true,
if it has more it will send tokens to its parent (state S2.4) until it remains Avg
tokens.
Induction. If the subtree has height n + 1 it has subtrees of height

\[^{0}\text{each neighbor of } v_j \text{ has a different value for } \text{subtreeAvg}(j) \text{ but, being the tree rooted, we refer the value owned by the neighbor } v_{\text{parent}} \text{ which is the parent of } v_j.\]
Fig. 1. Example of execution of our algorithm. Dashed arrows represent information messages, solid arrows represent token exchanges. (a) Phase 1: local information is sent from leaves to a single node which becomes the root of the tree. (b) Phase 2: token are redistributed in subtrees until each subtree has Avg number of tokens. (c) End of Phase 2: each node has Avg tokens but the root that may contain some extra token. (d) End of Phase 3: a perfect token distribution is achieved and each node receives a Finished message.

Fig. 2. State transitions diagram of phase 1.

n. Since the algorithm behaves exactly in the same way for every node in the tree except the root we can apply induction hypothesis on them. When subTreeBalanced = true the root of the subtree has at least Avg tokens and sends extra tokens toward the root of the tree as in the base case (state S2.4) and every other node finishes with Avg tokens for induction hypothesis, so the lemma holds. If subTreeBalanced = false for induction hypothesis all subsu-
trees with \( \text{subTreeAvg} \geq \text{Avg} \) will have \( \text{Avg} \) tokens in each node and send extra tokens to the root of the subtree. The root will receive such tokens (states S2.1 and S2.3) and, since the total number of tokens of the subtree is at least \( \text{subTreeN}[j] \cdot \text{Avg} \), will eventually have at least \( \text{Avg} \) tokens. If there are subsubtrees with \( \text{subTreeAvg} < \text{Avg} \) their root can not be in state S2.4 and so will receive tokens from their parent. Their parent, the root of the subTree, has more than \( \text{Avg} \) tokens and sends some of them to such subsubtrees (state S2.3). Reasoning again on number of tokens the root will make such subsubtrees to have \( \text{subTreeAvg} = \text{Avg} \) remaining at least \( \text{Avg} \) tokens. At this point \( \text{subTreeBalanced} \) will become true and the lemma holds, as explained above.

\[ \square \]

**Theorem 2.** At the end of phase 2 the root of the tree has at least \( \text{Avg} \) tokens while all the other nodes have exactly \( \text{Avg} \) tokens.

**Proof.** Consider \( \text{subTreeBalanced} = \text{true} \) at the root of the tree. Then we apply lemma 1 to all the children \( v_j \) of the root: all nodes in their subtrees have exactly \( \text{Avg} \) tokens. Then, for definition of \( \text{Avg} \), the root has at least \( \text{Avg} \) tokens and the theorem holds. Consider \( \text{subTreeBalanced} = \text{false} \) at the root of the tree. Then we apply lemma 1 to all the children \( v_j \) of the root having \( \text{subTreeAvg}(j) \geq \text{Avg} \). During phase 2 the root receives tokens (states S2.1 and S2.3) from subtrees having \( \text{subTreeAvg}(j) > \text{Avg} \). Since the total number of tokens of the tree is at least \( N \cdot \text{Avg} \), when such subtrees reach balancing the root has at least \( \text{Avg} \) tokens. If there are subtrees having \( \text{subTreeAvg}(j) < \text{Avg} \) the root will have more than \( \text{Avg} \) tokens and will send some of them to such subsubtrees (state S2.3). The root \( v_j \) of such subtrees can not be in state S2.4 and then receives tokens sent from the root. Reasoning again on number of tokens the root will make such subsubtrees to have \( \text{subTreeAvg}(j) = \text{Avg} \) remaining at least \( \text{Avg} \) tokens. At this point \( \text{subTreeBalanced} \) will become true. We apply lemma 1 to all the children \( v_j \) of the root and the theorem holds.

\[ \square \]

### 4.3 Phase 3.

**Theorem 3.** After phase 3 each node in the tree has \( \text{Avg} \) or \( \text{Avg} + 1 \) tokens.

**Proof.** We make the proof by induction on the height of the tree:

**Base case.** If the tree has height 0 (only one node) being the total number of tokens at most \( \text{Avg} + 1 \) the theorem is trivially true.

**Induction.** If the subtree has height \( n + 1 \) it has subtrees of height \( n \). If the root has at most \( \text{Avg} + 1 \) tokens the theorem is proved for theorem 2. If it has more tokens it sends them to its sons until it remains \( \text{Avg} + 1 \) tokens, due to the total number of tokens this will happen before that all subtrees have \( \text{subTreeAvg} = \text{Avg} + 1 \). At this time the root will send \( \text{Finished} \) to all its sons. On the other hand each son receives tokens without sending them and sets parent to 0 upon receiving the \( \text{Finished} \) message. Now the algorithm repeats recursively on each subtree rooted at each son of the root. Since the average number of tokens for each subtree is at most \( \text{Avg} + 1 \) the total number of tokens in the subtree is such that its root can send all extra tokens to sons before their
subtrees have more than Avg +1 average number of tokens. We can then apply
induction hypothesis to every subtree proving the theorem. □

5 Complexity.

We call $d$ the maximum degree of the tree, $T$ the total number of tokens, $D$ the
diameter of the tree and $N$ the number of nodes. Phase 1 runs in (worst case)
$O(Dd)$ in single-port model\footnote{Parameter $D$ in the model.}. Phase 2 runs in (worst case) $O(TD)$, while phase 3
runs in (worst case) $O(D(T \text{ mod } N))$. Since phase 2 has the greatest complexity
the overall complexity of the algorithm is $O(TD)$.

The space needed to store local information at each node is that for subtrees
load and number of nodes, plus a constant number of local data of size less
or equal to the elements of subtrees array, so size complexity at single node is
$\Theta(d \log T + \log N)$.

In phase 2 all tokens travelling through a link go in the same direction
and, if $\text{Avg} = T/N$, algorithm achieves perfect token distribution. This means
that the number of token exchanges is optimum. If $\text{Avg} < T/N$ then at most
$T \text{ mod } N$ tokens generate extra token exchanges. The longest path done by such
tokens is from a leaf to the root. So the number of token exchanges is at most
$O(D(T \text{ mod } N))$ greater than optimum. In phase 3, unless extra tokens were al-
ready at the root, we have already done more token exchanges than the optimum,
so all the $O(D(T \text{ mod } N))$ worst case token exchanges are beyond the optimum.
The total number of token exchanges is then no more than $O(D \min(T,N))$
greater than optimum.

6 Conclusions and Further Work.

We presented a distributed algorithm which computes a perfect token distribu-
tion on anonymous tree connected networks without taking as input any global
information about the tree structure or about the initial distribution of tokens.
Some other questions remain open and deserve further investigation. Among
them we wish to point out the following two that are strictly related to the
results presented in this paper.
(1) Each node in our algorithm has a local memory of size $\Theta(d \log T + \log N)$.
Is it possible to compute perfect token distributions on trees assuming that each
node has only constant size local storing capacity?
(2) Is it possible to compute perfect token distribution in a fully decentralized
way minimizing the number of token exchanges?

References

1. J. E. Boillat. Load balancing and Poisson equation in a graph. Concurrency: Prac-