TIME SYNCHRONIZATION IN WIRELESS SENSOR NETWORKS

A Thesis Submitted to
the Graduate School of Engineering and Sciences of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE

in Computer Software

by
Ali Burak KULAKLI

October 2008
İZMİR

We approve the thesis of **Ali Burak KULAKLI**

Prof. Dr. Sıtkı AYTAÇ
Supervisor

Prof. Dr. Kayhan ERCİYEŞ
Co-Supervisor

Assist. Prof. Dr. Aylin KANTARCI
Committee Member

Assist. Prof. Dr. Tolga AYAV
Committee Member

Dr. Belgin ERGENÇ
Committee Member

16 October 2008

Prof. Dr. Sıtkı AYTAÇ
Head of the Computer Engineering
Department

Prof. Dr. Hasan BÖKE
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGEMENTS

I would like to thank my co-supervisor, Prof. Dr. Kayhan Erciyeş, for his guidance and patience. He was my supervisor until he left the institute and he continued to support me as co-supervisor after.

Also, I would like to thank my supervisor, Prof. Dr. Sıtkı Aytaç, for his guidance after he became my supervisor.

Furthermore, I would like to thank my friends, especially Esra Aycan, helping me when writing this thesis using $\LaTeX$. Without them, it may have taken too much time to write this document.

Finally, I would like to thank my family who always supported me whole the time as well as my graduate education.

ABSTRACT

TIME SYNCHRONIZATION IN WIRELESS SENSOR NETWORKS

In this thesis, an enhanced synchronization algorithm for Wireless Sensor Networks is proposed. This algorithm uses TPSN (Timing-sync protocol for sensor networks) as a base synchronizer and does modifications on it to achieve a better synchronization with a lower message overhead. Basically, there are three improvements that can be applied onto TPSN, which are clustering the network, chain synchronization among nodes and adaptive synchronization interval. In the first phase of the thesis, a simulation environment is provided for TPSN using pthreads on a Linux computer. This environment helps understanding the parameters that TPSN relies on and testing the algorithm in different simulated environments with different characteristics using the enhancements onto TPSN algorithm. In the second phase, *ns2* simulator environment is used to get more precise results and test the modifications. Finally, latest modifications are done on TPSN and all the results are gathered from *ns2*.

ÖZET

KABLOSUZ DUYARGA AĞLARINDA ZAMAN EŞUYUMLULUĞU

Bu tezde amaç kablosuz duyarga ağları için geliştirilmiş bir zaman eşuyumluluğu algoritması sunmaktır. Bu algoritma temelde bir alıcı-verici tipinde algoritma olan TPSN'i kullanmakta olup, daha iyi bir zaman eşuyumluluğunu az bir mesaj yükü artışıyla sağlamak üzere değişiklikler yapılmıştır. Aslında, burada 3 adet iyileştirme bulunmaktadır ki bunlar ağın kümelendirilmesi, zincir halinde zaman eşuyumluluğunun sağlanması ve şartlara göre zaman eşuyumluluğu algoritmasının işlem aralığının değiştirilmesidir. Tezin birinci aşamasında bir Linux bilgisayar üzerinde pthread'ler kullanılarak bir simulasyon ortamı hazırlanmıştır. Bu ortam, TPSN'nin dayandığı parametreleri anlamamıza ve algoritmanın farklı ortamlar ile karakteristiklerde, TPSN üzerine yapılan geliştirmelerin de eklenmesiyle test edilmesine yardımcı olmuştur. İkinci aşamada, daha kesin sonuçlar edinmek ve geliştirmeleri test edebilmek için *ns2* simulasyon ortamı kullanılmıştır. Sonuçta, son değişiklikler TPSN üzerine uygulanmış ve *ns2* üzerinden tüm sonuçlar alınmıştır.

TABLE OF CONTENTS

LIST OF FIGURES viii

CHAPTER 1 . INTRODUCTION 1

CHAPTER 2 . BACKGROUND 4

2.1. Synchronization Problem 4

2.1.1. Clock Drift 4

2.1.2. Clock Offset 5

2.2. Synchronization Algorithms 6

2.3. Timing-sync Protocol for Sensor Networks (TPSN) 6

2.3.1. Pair-wise Synchronization 6

2.3.2. Network-wide Synchronization 7

CHAPTER 3 . ENHANCEMENTS 9

3.1. Clustering 9

3.1.1. Dividing into Clusters 9

3.1.1.1. Balance Children Count 12

3.1.1.2. Balance Cluster Node Count 15

3.1.2. Inter Cluster Synchronization 17

3.1.2.1. Using the Existing Spanning Tree 17

3.1.2.2. Building a New Path Among Cluster Heads 17

3.2. Chain Synchronization 18

3.3. Adaptive Synchronization Interval 20

3.3.1. Runtime or Boottime 21

CHAPTER 4 . SIMULATION 22

4.1. Simulation Using Posix Threads 22

4.2. Simulation Using *ns2* 23

4.2.1 . Reference TPSN Results 24
4.2.2 . Enhancement Results 28
4.2.2.1 . Clustering 28
4.2.2.2 . Chain Synchronization 30
4.2.2.3 . Adaptive Synchronization Interval 32
4.2.2.4 . Balanced Children Count 34
4.2.2.5 . Balanced Cluster Node Count 35

CHAPTER 5 . CONCLUSION 37

REFERENCES 41

APPENDICES

APPENDIX A. SIMULATION USING POSIX THREADS (SOURCE CODE) . 42
APPENDIX B. SIMULATION USING NS2 (SOURCE CODE) 46

LIST OF FIGURES

Figure **Page**

Figure 3.1. Clustered Spanning Tree 10

Figure 3.2. Algorithm To Form Clusters 11

Figure 3.3. Dense Network Spanning Tree 11

Figure 3.4. Balanced Dense Network Spanning Tree 12

Figure 3.5. Algorithm To Balance Children Count 13

Figure 3.6. Balanced Children Spanning Tree 14

Figure 3.7. Algorithm To Balance Cluster Node Count 15

Figure 3.8. Balanced Clustered Spanning Tree 16

Figure 3.9. Spanning Tree For Chain Synchronization Example 18

Figure 3.10. Algorithm To Adjust Synchronization Interval 21

Figure 4.1. Clock Drifts Of Nodes 23

Figure 4.2. TPSN Spanning Tree 24

Figure 4.3. Spanning Tree Construction Time With Level Discovery 25

Figure 4.4. Spanning Tree Construction Time(Init) With Level Request 25

Figure 4.5. Spanning Tree Construction Time(Join) With Level Request 26

Figure 4.6. Clock Drifts Of Nodes (Reference TPSN) 27

Figure 4.7. Clock Difference Of Nodes (Reference TPSN) 27

Figure 4.8. Clock Difference Of Nodes (Reference TPSN) 28

Figure 4.9. Clock Difference Of Nodes (Clustering) 29

Figure 4.10. Message Complexity (Clustered TPSN) 29

Figure 4.11. Clock Difference Of Nodes (Chained TPSN) 30

Figure 4.12. Message Complexity (Chain Synchronization) 31

Figure 4.13. Adaptive Synchronization Interval Change 32

Figure 4.14. Clock Difference Of Nodes (Adaptive Synchronization Interval) . . 33

Figure 4.15. Message Complexity (Adaptive Synchronization) 33

Figure 4.16. Clock Difference Of Nodes (Balanced Children) 34

Figure 4.17. Message Complexity (Balanced Children) 35
Figure 4.18. Clock Difference Of Nodes (Balanced Cluster) 36
Figure 4.19. Message Complexity (Balanced Cluster) 36
Figure 5.1. Individual Results Of The Enhancements 37
Figure 5.2. Cumulative Results Of The Enhancements 38

CHAPTER 1

INTRODUCTION

Wireless Sensor Networks(WSN) are large scale networks of sensors running on wireless environment dedicated to observe physical world (Sundararaman, et al. 2005). For an application running on a WSN, gathered data by the sensors are time critical at most of the cases. Synchronization is crucial to many sensor network applications that require precise mapping of the collected data from the sensors with the time information such as in tracking and surveillance (Li and Rus 2006). However, almost all the nodes suffer from a problem named clock drift and skew. That problem causes clock difference among nodes as time goes because the processors do not run at exactly the same speed. That will possibly cause application errors eventually because the input data is incorrectly timestamped and they will not be interpreted correctly by the application. For the time critical applications running on WSN, clock drift problem should be reduced to a reasonable level or completely eliminated if possible. Chapter 2 gives basic information about synchronization problems and current possible solutions.

A network does have at least two nodes and all the nodes have their internal oscillators inside. Although they might have same frequencies as labelled, they may not be running at exactly same speed. This tiny difference cumulates as time goes and after a while, if there is no interference occured to synchronize the nodes, clocks will have significant differences. Section 2.1.1 describes the synchronization problem and the clock drift. In addition to the clock drift, some nodes might be started later then the other nodes in the network or some can be added to the network later. In such cases, there would be a fixed clock difference offset. Section 2.1.2 describes the synchronization problem and the clock offset.

With the emergence of WSNs, current LAN synchronization methods will not work efficiently. For example, although GPS provides good synchronization accuracy, it requires a very large amount of power. NTP is also infeasible since it is designed for traditional computer networks and will not scale well for wireless sensor networks and requires processing power (Saravanos 2006). Sensor nodes are very tiny instruments and running with a limited energy so it is not easy to synchronize nodes effectively because of energy consumption. Also, they usually do not have much processing power and this prevents any complex algorithms to run on WSN. There are some other synchronization methods specifically for sensor networks which are described in Section 2.2 such as the Timing-sync Protocol for Sensor Networks (TPSN) and the Reference Broadcast Synchronization method (RBS). RBS is a receiver-receiver based protocol. On the other hand, TPSN is a sender-receiver based protocol. A dedicated Section 2.3 gives detailed information about TPSN because it is the base algorithm that will have enhancements onto it.

Chapter 3 describes the proposed modifications to have a better synchronization. First modification is clustering the network, which is shown in Section 3.1. This will reduce complexity of the network and lets us have parallel synchronization where one of them is the inside cluster synchronization and the other one is the inter cluster synchronization. That means, different synchronization algorithms can be used inside and inter cluster synchronization. However, TPSN is used here for both inside and inter cluster synchronization with different parameters. Clustering does have two extra enhancements inside which are balancing children count in Section 3.1.1.1 and balancing cluster node count in Section 3.1.1.2. Another enhancement is doing synchronization in chains when available and updating clocks of more than one node in a synchronization state. This is described in Section 3.2. Last enhancement is given in Section 3.3 which is using different but optimized synchronization intervals for each node. This will bring a balanced message load on the network and possibly increase precision.

Simulation is the critical part of this algorithm optimization and enhancements so Chapter 4 gives detailed information about the simulation environments and the results of the enhancements. During the preparation of this thesis, two simulations are done independently. First one is a new simulation environment built from scratch that lets test the TPSN behaviour and do simple changes on it. Details of this simulation can be found in Section 4.1 however, the results from this simulation may not be considered as a reliable source for the effects of the modifications onto TPSN so well-known *ns2* environment is used primarily to test the enhancements. Section 4.2 has all the implementation information and the results gathered from *ns2*. All the other details are available in Appendix section.

CHAPTER 2

BACKGROUND

2.1. Synchronization Problem

Time synchronization problem consists of giving all the nodes of the network a common time scale to operate. This common time scale is usually achieved by periodically synchronizing the clock at each node to a reference time source, therefore the local time seen by each element of the system is approximately the same. Also, time synchronization plays an important role in wireless sensor networks because it allows the entire system to cooperate (Hu and Servetto 2005).

The differences on the clocks of sensor nodes at any time is referred as the offset error between them. There are three reasons for the nodes to be representing different times in their respective clocks. The nodes might be started at different times as described in Section 2.1.2, the quartz crystals at each of these nodes might be running at slightly different frequencies, causing the clock values to gradually diverge (clock *skew*) or the frequency of the clocks can change variably over time because of environment conditions such as temperature (clock *drift*) (Ganeriwal, et al. 2005a). Section 2.1.1 gives the details about clock drift and skew problem. In this document, term *skew* will not be used and it should be considered in *drift*.

2.1.1. Clock Drift

Let a wireless network has K sensors, where each node has a discrete-time clock with period T_k. If the nodes are isolated, the timing clock of the kth sensor evolves as

$$t_k(n) = nT_k + \tau_k(0) \tag{2.1}$$

where $0 \leq \tau_k\ (0) < T_k$ is an initial arbitrary phase and n = 1, 2, ... runs over the periods of the timing signal (Simeone and Spagnolini 2007).

Each sensor has a clock generator crystal in it and those crystals generates a clock for the sensor and the period is T_k. Although all the sensors have labelled as the same working frequency, they all have a margin of errors. That means they may not be running at exactly same speed. That difference then causes a clock drift. As a result, after some time, although all the sensors are started exactly the same time, they may not be at the same clock. Basically, the clock drift is a hardware problem caused by variation in the crystal frequency due to noise, temperature, aging, voltage change etc. and the cost of the crystal increases with the accuracy and the low-cost nodes in the sensor network generally use less accurate crystals which has larger margin of errors (Tjoa, et al. 2004). This is the main synchronization problem because most of the times a sensor data with a time data provides an information for an application. That is, without a correct time information, data is not valuable and even it might affect application's accuracy.

2.1.2. Clock Offset

When there were no clock drift problem, all network nodes would be started at the same time to have no clock difference in the network. If not, there would be a fixed clock difference between nodes and this difference must be handled by the application (Ganeriwal, et al. 2005a). Assume that a node starts at $n = 0$ and another node starts at $n = 5$. If those nodes have exactly same periods (T), there would be a fixed clock difference between two nodes and the formula 2.1 becomes

$$\Delta(t_2 - t_1) = 5T + \tau_2 - \tau_1 \tag{2.2}$$

The $5T$ here is the clock offset must be handled with a synchronization algorithm. Also, it is obvious that, even all the nodes start at exactly the same time (n), there will be a tiny fixed clock offset, which is $\Delta(t_2 - t_1) = \tau_2 - \tau_1$.

2.2. Synchronization Algorithms

There are several time synchronization protocols because of varying requirements, such as precision or degree of mobility. Time synchronization procedure typically is a message exchange containing the timestamp and the measurement of delay. There are three basic solutions for time synchronization in sensor networks (Chen, et al. 2007):

1. Sender-Receiver Based Synchronization
2. Receiver-Receiver Based Synchronization
3. Delay Measurement Synchronization

Receiver-Receiver based synchronization algorithms use one-way message exchange commonly such as in the Reference Broadcast Synchronization (RBS) (Elson, et al. 2002). On the other hand, two-way message exchange is used in Sender-Receiver based synchronization protocols, such as the Timing-sync Protocol for Sensor Networks (TPSN) (Ganeriwal, et al. 2003). There are also some synchronization protocols based on one-way message exchange as well as the measurement of delay. An example of a such protocol is Delay Measurement Time Synchronization (DMTS) (Ping 2003).

2.3. Timing-sync Protocol for Sensor Networks (TPSN)

TPSN is a Sender-Receiver based time synchronization protocol for wireless sensor networks. It has two main steps to synchronize the network. Firstly, pair-wise synchronization will be explained and then network-wide synchronization.

2.3.1. Pair-wise Synchronization

Let's have 2 nodes i and j which will be synchronized and node i starts the synchronization. Here are the steps to synchronize node i to node j:

1. Node *i* creates a synchronization pulse packet and hands over packet to the operating system and network stack for transmission.
2. Just before transmission, the packet is timestamped with T_1 and delivered to the medium. This prevents uncertainties of network stack and medium access delay.
3. After propagation delay and packet transmission time, the packet will be delivered to node j. Then it is timestamped with T_2 immediately to prevent uncertainty again.
4. Node *j* then prepares a synchronization acknowledgment packet and hands it over to the operating system and network stack.
5. Like in step 2, just before transmission, the packet is timestamped with T_3 and delivered.
6. Node *i* receives the packet and timestamps again with T_4 for last time.
7. Finally, node *i* calculates the clock offset and fixes its clock.

The calculation is simple and can be computed very fast by the nodes. Let's say there is an offset O and it needs to be calculated when reply package comes. Assuming there is no timestamping uncertainties, the formula is just like below (Ganeriwal, et al. 2003):

$$O = \frac{(T_2 - T_1) - (T_4 - T_3)}{2} \tag{2.3}$$

After computing the offset, the clock difference is known between node *i* and node j. It depends on the implementation to change local clock using the offset or keep the offset in a separate place and use it when needed.

2.3.2. Network-wide Synchronization

TPSN builds a spanning tree where each node knows the level of itself and the parent. Level 0 is assigned to one node which is named as root node. This node has the responsibility to build tree by triggering level discovery phase. To start the tree construction, root node sends a level discovery packet with its level 0 in it. When all the one hop neighbors receive this packet, they set their level to 1, parent to 0 and send another level discovery packet with the level 1 in it. However, before sending another packet, each node waits for random time to avoid collisions. This process is done for all other nodes. In summary, when a node receives a level discovery packet, it sets its level to the level one more in the packet and sets the sender node as its parent.

What if a node fails to receive level discovery packet or a node is added to a network later? After a period of time, if a node could not get any level discovery packet, it sends a level request packet. This also occurs when sync requests fail. When its neighbors receive this level request message, they reply the message with their level in it. Then the node sets its level to the minimum received level + 1 and sets it parent to the sender of minimum level.

Synchronization is done for each node periodically. Each node sends a sync pulse packet to its parent periodically to be synchronized. When a node cannot get a synchronization acknowledgment packet from its parent, that may mean that its parent is dead therefore it sends a level request packet to set a new level itself (Ganeriwal, et al. 2003).

CHAPTER 3

ENHANCEMENTS

3.1. Clustering

In this method, a spanning tree which is clustered using a level depth is used. So, there will be clusters in the network. All nodes except cluster roots will keep running the existing synchronization algorithm to synchronize their clocks however, another way will be introduced to synchronize cluster roots in parallel. If this can be achieved, there will be a better synchronized network. Because, pure TPSN has a problem that when the level of a node increases, the clock difference may also increase. With clustering, it is possible to decrease levels of the nodes and with a good inter cluster synchronization, a better synchronized network can be available even using pure TPSN without the other enhancements described here.

3.1.1. Dividing into Clusters

There is a directed spanning tree that TPSN algorithm generates. In order to divide the network into clusters, node level parameter in TPSN is used and a similar level parameter is introduced for clusters. Therefore, each node will have a level and a cluster (or group) level. With a reasonable fixed level of nodes for clusters, next level nodes in the cluster will be in the next cluster and levels of the nodes are reset to 0 again but the cluster level is increased by one. Figure 3.2 shows the level setting algorithm when clustering is enabled. When 4 is used as $maxLevelInClusters$ for each cluster, the reference spanning tree becomes just like in Figure 3.1.

If the message complexity of this clustering algorithm is considered as the number of messages exchanged between nodes until a valid clustering structure is achieved, it has the same complexity to build a spanning tree, which is $O(n^2)$ where n is the number of nodes. The reason behind this complexity is the message broadcast at level discovery phase. In a very dense network, a node may get $(n-1)$ messages from other nodes and there are n nodes, so the message complexity becomes $O(n^2)$.

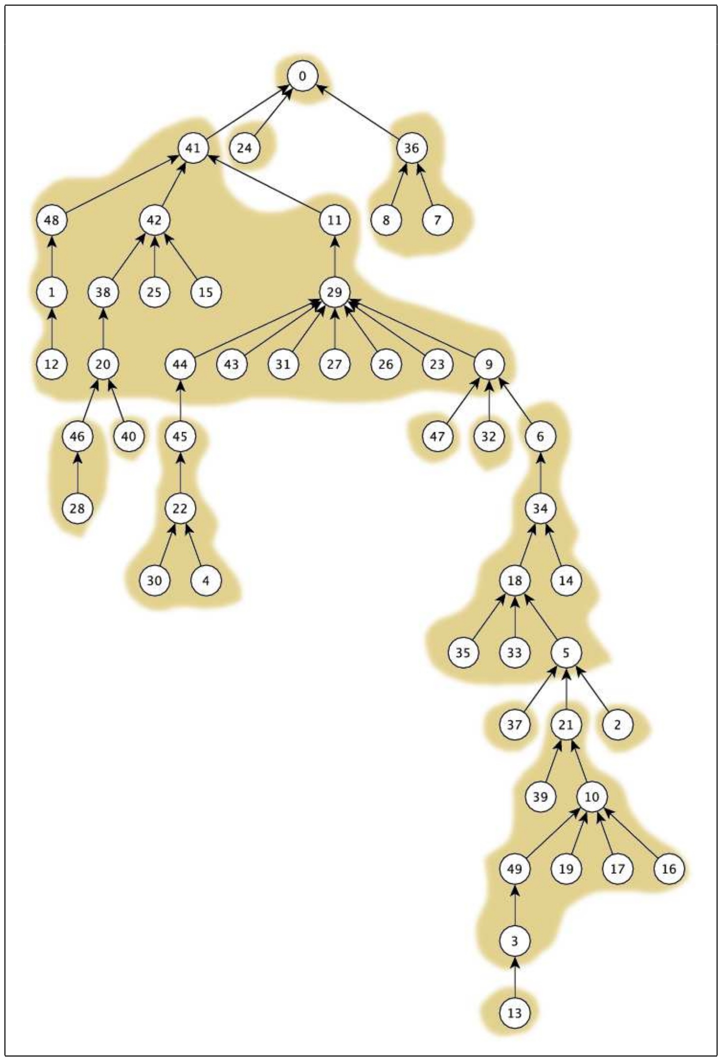

Figure 3.1. Clustered Spanning Tree

On the other hand, if the time complexity of this clustering algorithm is considered the total time required to form a valid clustering, the time complexity becomes $O(n)$ because a node can wait at most all other nodes to be initialised one by one.

However, such simple no-overheaded clustering might not produce balanced clusters. Consider a dense network which all the nodes are very close to each other. Then, it is likely that the spanning tree will have only two levels which level 0 is the root and all the children are at level 1 as shown in Figure 3.3. In that case, it is not possible to use clustering enhancement effectively on such spanning tree.

```
input : $parentNodeLevel$, $parentNodeClusterLevel$, $maxLevelInClusters$ where
        $parentNodeLevel \geq 0$, $maxLevelInClusters \geq 1$ and
        $parentNodeClusterLevel \geq 0$.

  $candinateNodeLevel = parentNodeLevel + 1$;
  if $candinateNodeLevel < maxLevelInClusters$ then
      $nodeLevel = candidateNodeLevel$;
      $nodeClusterLevel = parentClusterLevel$;
  else
      $nodeLevel = 0$;
      $nodeClusterLevel = parentClusterLevel + 1$;
  end
```

Figure 3.2. Algorithm To Form Clusters

In another scenario, although there may be enough levels to divide a network into clusters, that clustering might make clusters whose number of nodes are not balanced. As shown in Figure 3.1, node 41 is a root of a cluster which has 18 nodes however, there are a couple of clusters that has only 1 node such as 47.

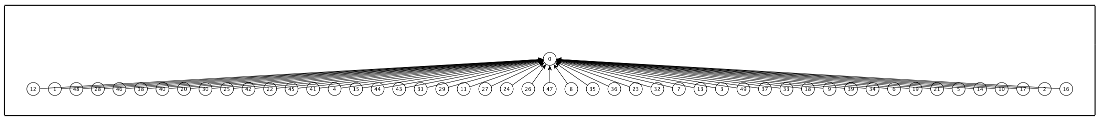

Figure 3.3. Dense Network Spanning Tree

That is why two other enhancements are introduced to provide mored balanced clusters. First enhancement balances children count of a parent and second enhancement balances number of nodes in a cluster. These enhancements use existing TPSN synchronization messages and run over the initialized spanning tree. That means, they provide dynamic balancing after initialization phase.

3.1.1.1. Balance Children Count

In simple TPSN spanning tree construction in dense networks, an unexpected type of spanning tree can be constructed which a node might have many children as shown in Figure 3.3. This is not proper because the parent consumes too much energy and lasts shorter because of replying synchronization request messages. This is a downside of current method and should be fixed.

In order to balance children count, existing TPSN synchronization messages can be used, that means no extra message overhead will occur. However, each node must store their children information to meet the requirements. That also means, if at most 4 children is expected for a node, each node will reply requests from only 4 of their children and other children will eventually disconnect and find a new parent.

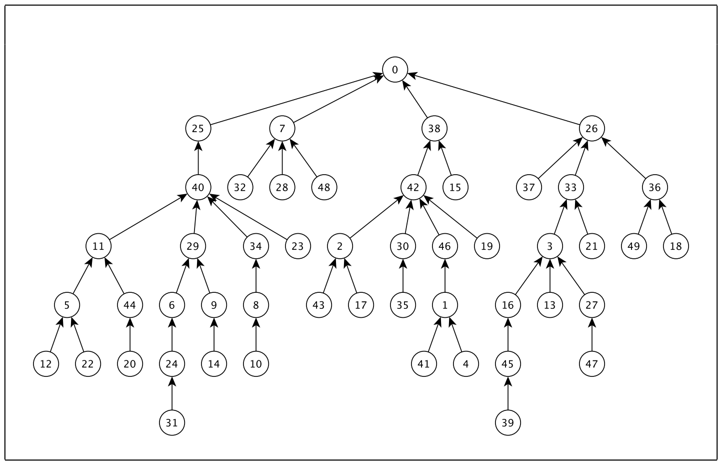

Figure 3.4. Balanced Dense Network Spanning Tree

At first, each node does not know any of their children. However, when synchronization starts, each node starts to get requests from their children. Therefore, as algorithm in Figure 3.5 shows, if a node requests a synchronization from its parent, the parent checks the number of children, which is 0 at the beginning, and if the number of children is less than maximum number of children allowed, then the parent adds the child to its children list and replies the request. On the other hand, if the parent already has maximum allowed number of children, then it replies the request but this reply message is different and causes child node to disconnect from its parent. In this situation, not replying the request is also reasonable but not recommended because it might decrease precision of synchronization.

The disconnection then causes the child to request a new parent. A short time after initialization, there will be a balanced network. Figure 3.4 shows how a dense network spanning tree is changing dynamically where *maxChildrenCount* parameter is used as 4.

```
input : $activeChildrenCount$, $maxChildrenCount$ where $maxChildrenCount \geq 1$,
        $activeChildrenCount \geq 0$.

if $activeChildrenCount < maxChildrenCount$ then
    add child to list;
    ++$activeChildrenCount$;
    reply sync request as usual;
else
    reply sync request with disconnect request;
end
```

Figure 3.5. Algorithm To Balance Children Count

This obviously increases time complexity of spanning tree construction however, this is related to the fact that we use existing TPSN pair-wise synchronization messages to update spanning tree. Also, this enhancement does not need to be run at initialization, which means this is actually not increasing the time of spanning tree construction. Time synchronization may continue on some nodes during this balancing process. Adding an extra message to say child node to disconnect immediately does not increase message complexity because, otherwise, child node keeps trying to synchronize for a while up to the implementation which will definitely increase the messages required to balance the network.

One important issue here to handle is keeping active children list up to date. So, a timeout should be enabled for each children and each request should reset that timeout. When a child is timeouted, it can be safely removed from the list. However, it is possible to add that node again to the list and such will not affect the synchronization results because these are all done in parent node, that means no message exchange is required.

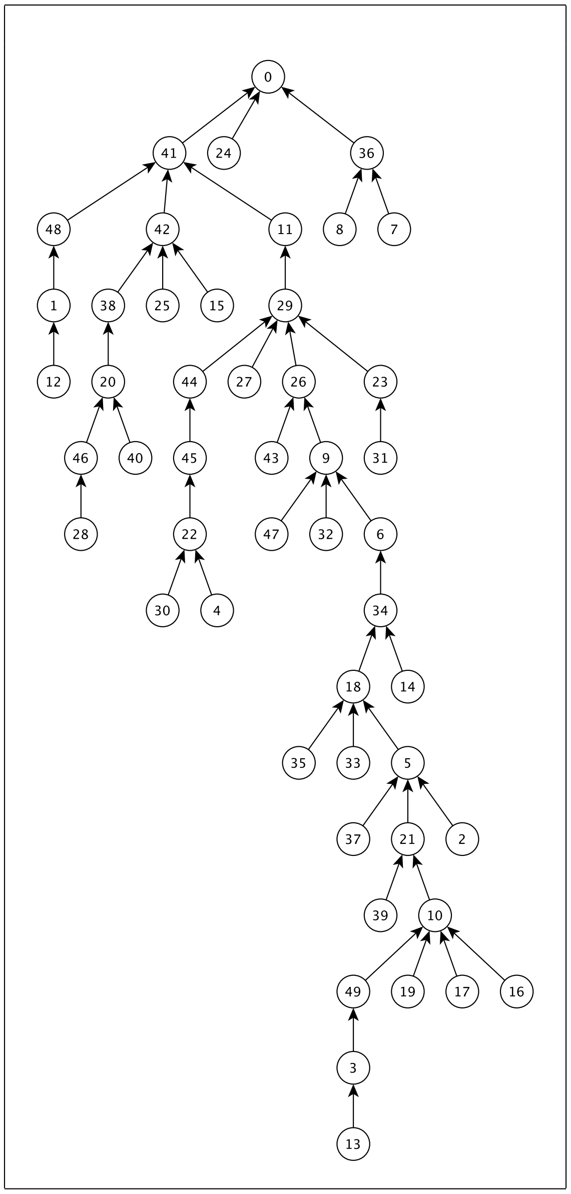

Figure 3.6. Balanced Children Spanning Tree

Figure 3.6 shows the spanning tree after balancing the children count and node 29 has less children now. *maxChildrenCount* parameter is used as 4 here.

3.1.1.2. Balance Cluster Node Count

Clustering is dividing the network to distribute synchronization work. However, the clustering algorithm just sets a group level to a node using its level parameter in TPSN and it can be seen in Figure 3.1, it is highly likely to have very big and very small clusters in a network. If more balanced clusters exist, then a better message complexity can be possible.

```
input : $descendantCount$, $otherDescendantCount$, $maxNumberOfNodesInCluster$,
        $currentLevelInCluster$, $maxLevelInCluster$ where $descendantCount \geq 0$,
        $otherDescendantCount \geq 0$, $maxNumberOfNodesInCluster \geq 1$,
        $currentLevelInCluster \geq 0$, $maxLevelInCluster \geq 1$.

if $otherDescendantCount >= maxNumberOfNodesInCluster$ and
$descendantCount >= maxNumberOfNodesInCluster$ then
    disconnect node;
else
    if $currentLevelInCluster < maxLevelInCluster$ and ($descendantCount == 0$
    or ($otherDescendantCount < maxNumberOfNodesInCluster$ and
    $descendantCount > maxNumberOfNodesInCluster$)) then
        join node;
    end
end
```

Figure 3.7. Algorithm To Balance Cluster Node Count

This enhancement is a dynamic clustering algorithm again using TPSN pair-wise synchronization messages. If a $maxNumberOfNodesInCluster$ is provided, then the algorithm calculates descendant counts using the children count calculated in Section 3.1.1.1. That means, each node will know its descendant count which root node will have $numberOfNodes - 1$.

The algorithm simply disconnects nodes from and joins nodes to a cluster using a simple decision algorithm. That says, if descendant count of a node except node to disconnect is enough, then disconnect that node. If descendant count of a node except node to join is not enough, then join that node. Algorithm in Figure 3.7 shows the details.

This algorithm is dependent to the *maxLevelInCluster* parameter. Therefore, joining may not be possible because of the *maxLevelInCluster* parameter. This disconnect/check condition can be improved or changed according to environment.

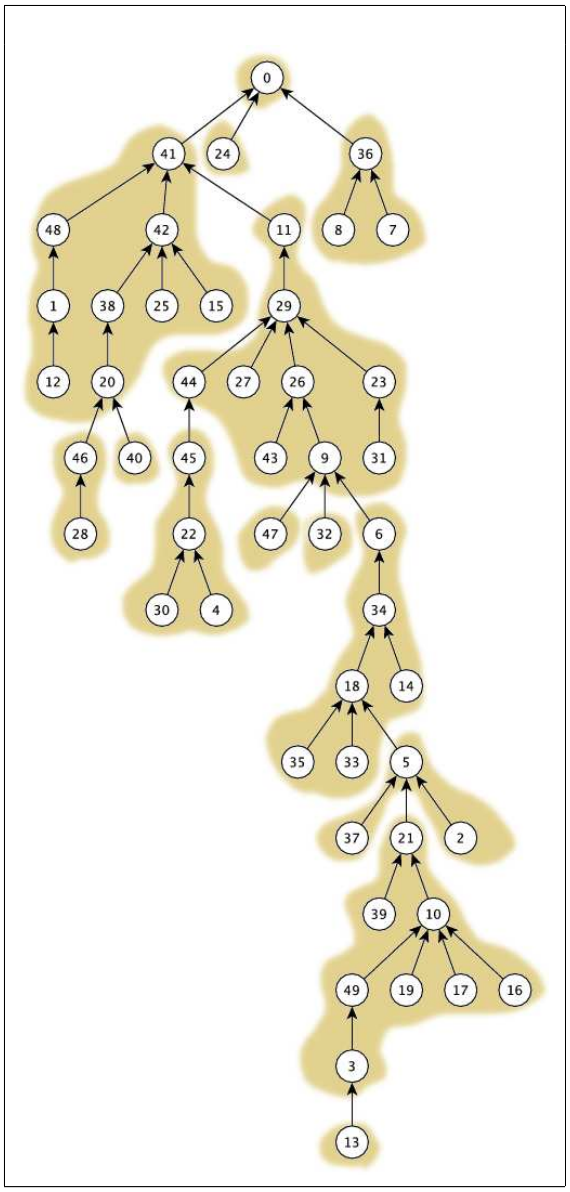

Figure 3.8. Balanced Clustered Spanning Tree

Figure 3.8 shows the change in clustering after balancing. *maxLevelInCluster* parameter is 4 at this figure and it is run with the balanced children enhancement described in Section 3.1.1.1.

3.1.2. Inter Cluster Synchronization

When the network spanning tree is available with the clusters, now it is required to find a way to synchronize cluster roots. There are two ways to achieve this. First is using the current available path to synchronize, second is building a new path among cluster heads.

3.1.2.1. Using the Existing Spanning Tree

There is already a spanning tree for clusters and a path is available among between all child cluster roots to their parent cluster roots. That means, it is possible to use that path to send and receive packets to synchronize cluster roots. This can be effective if the levels in cluster are not many because, from child cluster root node to parent cluster root node, there will be level number of hops which will increase message count. Therefore, with a reasonable level count in a cluster, this method will work well enough to improve synchronization in the network. The method described in 3.2 can be used to synchronize cluster heads. The idea in this enhancement is not doing as much synchronization requests as in the cluster. For instance, if a cluster node requests synchronization for every 100s, cluster root requests for every 200s. Getting similar synchronization results with less message exchange in the network is expected here. In Section 4.2, simulation results of this behavior can be found.

3.1.2.2. Building a New Path Among Cluster Heads

Another way of doing inter cluster synchronization is providing another path among cluster root nodes which can be considered another process running on the same tree in parallel. In this method, a better synchronization than using existing tree as described in 3.1.2.1 can be achieved and required messages will be decreased because hop counts for the messages will be less. However, this increases application complexity. In this thesis, existing spanning tree is used to show there is no reason to make another path between clusters.

3.2. Chain Synchronization

In pure TPSN, only the child and its parent are involved to synchronize the child's clock. In this proposed method, at some convenient cases, it should be possible to synchronize child clock to any of the parent nodes even the root node of the tree or the cluster root if clustering is done as described in 3.1.2. This will increase messages required but it also increases the precision.

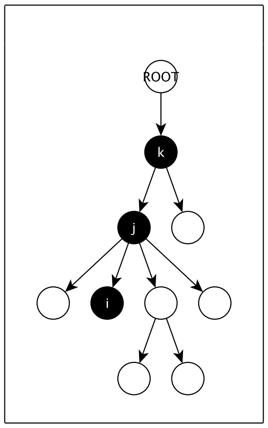

Figure 3.9. Spanning Tree For Chain Synchronization Example

Here is a simple scenario for the network shown in Figure 3.9 to understand how chain synchronization works. In this scenario, Node *i* is the child, requesting synchronization. Node *j* is the intermediate node and node *k* is the root to be synchronized.

When node *i* starts the synchronization, it asks for the time to its parent, which is node *j*. TPSN says, Node *j* answers this request packet with acknowledge packet however, this time it forwards the synchronization request to node *k*. Before forwarding the request packet, node *j* has 3 important jobs to do.

1. *Store requester node address*: Because TPSN tree structure is directed and only the children know their parent. Therefore node *j* stores node *i* as requester node.
2. *Store requester sent time*: This is crucial for TPSN offset calculation and it will be

used when sending acknowledge packet to node *i*.

3. *Replace requester sent time in the packet*: To update intermediate node *j*'s clock, a new TPSN packet must be prepared. Therefore, requester sent time information is replaced with the node *j*'s sent time as TPSN requires.

When node *k* receives the forwarded packet, it is just a TPSN packet from node *j*. So, node *k* replies the request with its time information. When node *j* receives the packet, this packet can be used to update node *j*'s clock and so, node *j* updates it clock. However, after updating the clock, node *j* replaces requester sent time information with the original one, which is stored before forwarding the request and forwards acknowledge packet to node *i*. With this packet, node *i* updates its clock correctly.

The assumption here is that TPSN does only use 4 timestamps to synchronize the clock so the time passes on the parent node logically is not important. It is easy to see that node *j*'s request forwarding and acknowledge waiting times can be considered as node *j*'s internal process. With this way of thinking, what is done here is actually nothing different than TPSN.

This operation can be done for all nodes but nodes with low level will be too busy for synchronization in that case, especially the root. That might reduce precision because they cannot answer synchronization requests when doing another chain synchronization. It is possible to do chain synchronization for all request by implementation but that is not acceptable because it will also increase message traffic drastically. Therefore, a simple solution may prevent this problem.

If a node is in chain synchronization and cannot answer child node's chain synchronization request, then it can reply request just like in TPSN. No chain request is required. TPSN like behaviour is kept here when an intermediate node is busy with chain synchronization. As another exception, if a node has just synchronized it's time, then there is no reason to forward the packet. It can just break the chain and reply the request immediately. Such decisions will absolutely decrease message traffic.

The chain synchronization here is actually a multi-hop TPSN with extra break decisions are added. That means, when a node *i* is being synchronized, it may do different count of hops for the synchronization and this depends to its parent's state as described above. That is how chain synchronization increases the precision without increasing message traffic significantly. The exception here is the inter cluster synchronization. That synchronization must be done between cluster roots whenever possible and no extra chain break conditions should done during the inter cluster synchronization.

Initially, this chain synchronization was suggested for inter cluster synchronization between cluster roots as Section 3.1.2 describes however, because of the fact that no different synchronization algorithms are used inside the clusters, which is possible by the way, this enhancement can be used with those break conditions in the cluster only synchronizations to improve inside cluster synchronization. The simulation results can be seen in Section 4.2.

3.3. Adaptive Synchronization Interval

In most cases, a clock difference up to some value is acceptable. If such a value is available or predictable, than there is a good option to use. Some nodes have bigger drifts and some nodes have smaller. This can also be changed after initialization of the network. So, it is not an optimized solution to use same intervals for all the nodes and initializing interval at the beginning. Gradually increasing or decreasing the interval using the last synchronization results may give better results. Small steps such as 1% is reasonable. After some time, there will be a network which is optimized for message load. It is obvious that, if the expected value for clock difference is kept small, than it will increase the message count. So, expected or the acceptable clock difference should be carefully selected. Algorithm in Figure 3.10 shows how adaptive synchronization works. As an exception, an allowed maximum change in interval value can be used such as 20%. So, the adaptive interval change will not go below 80% or over 120%.

The goal here is balancing message traffic and getting more precise results. Therefore, if a node does not have a high drift rate, there is no reason to synchronize it at same frequency with a high drift rated node. With selecting an adaptive synchronization interval, the message traffic can be reduced while increasing the precision.

```
input  : $currentInterval$, $currentClockOffset$, $expectedClockOffset$ where
         $currentInterval \geq 0$, $currentClockOffset \geq 0$, $expectedClockOffset \geq 0$.

  $candinateNodeLevel = parentNodeLevel + 1$;
  if $currentClockOffset < expectedClockOffset$ then
      $currentInterval = currentInterval * (101/100)$;
  else
      if $currentClockOffset > expectedClockOffset$ then
          $currentInterval = currentInterval * (99/100)$;
      else
          // No change in $currentInterval$ is required.
      end
  end
```

Figure 3.10. Algorithm To Adjust Synchronization Interval

3.3.1. Runtime or Boottime

It is possible to set different synchronization intervals during boottime by analyzing clock drift of the node. This might reduce the message traffic if handled correctly but it is not guaranteed to improve precision all the time. Because clock drift is not fixed. It depends on the environment and might be changed during node's life (Ganeriwal, et al. 2005b). Therefore, updating the interval during runtime with tiny steps will give the best results.

CHAPTER 4

SIMULATION

Time synchronization is an important service of wireless sensor networks. As proposed here, researchers have already proposed some other time synchronization algorithms and existing time synchronization algorithms often need to be adapted. Software simulation is a valid and quick way to evaluate these algorithms before using them (Chao-Nong, et al. 2006). In this chapter, information is given about the custom wireless network simulation for TPSN in Section 4.1 and the *ns2* implementation to test the enhanced time synchronization algorithm in Section 4.2.

4.1. Simulation Using Posix Threads

At first, a simple simulation environment is implemented using threads for each node. The idea behind this is understanding the behaviour of TPSN and implementing the protocol simply before migrating the enhancements described in Chapter 3 and applying some modifications over it to discover the effects of the parameters.

What is needed initially is a clock drifting simulation. That means, a way is required to create a clock drift for nodes. In order to achieve this, counters are used as node clocks and created a global thread to update counters. This global thread should increase some counters more than the others. In order to create a drift, uniform distribution is used. In the final simulation results, which is done in *ns2* described in Section 4.2, more suitable distributions are used as required. This simulation is actually done to get basic ideas about the time synchronization.

The global thread first resets all the clocks to 0 then increases counters for each node. It also increases counters again with a simple way (by creating a random drift rate) to create a drift. There is also a wrap point, which can be changed as required. Wrap point can be changed if the clock counter is 16 bit, 32 bit or any other.

For each node, a thread is used to handle the implementation of the node. This thread waits for TPSN messages and replies to them as TPSN requires. IPC message queue routines are used to send/receive messages among threads. Then TPSN behaviour is experimented and the enhancements over it on this new simulator to understand how it synchronizes the network. The source code is available in Appendix A. No other details are given here because results are needed from a well known simulator to show our enhancements effects.

4.2. Simulation Using *ns2*

In order to analyze enhancements done over TPSN, a simulation of a wireless sensor network with 50 nodes is prepared using *ns2 (v2.31)* simulator. All nodes are located in a 1000m x 1000m area randomly using uniform distribution. To generate clock drift for nodes, normal distribution is used with parameters μ=1 and σ^2=0.1. Simulation duration is 10000s and periodic interval is 100s. This will be used as a reference result before synchronization is done. Figure 4.1 shows drifts of the nodes when no synchronization is done. The divergence of the clocks here is obvious.

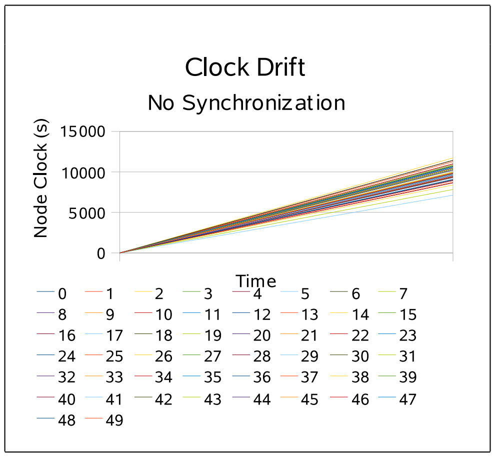

Figure 4.1. Clock Drifts Of Nodes

4.2.1. Reference TPSN Results

In this phase, TPSN is directly applied to the network. At first, level discovery phase is run. In Figure 4.2, the structure after level discovery phase can be seen. There are 14 levels in this network. Each row represents the level in TPSN protocol not the physical location of the node.

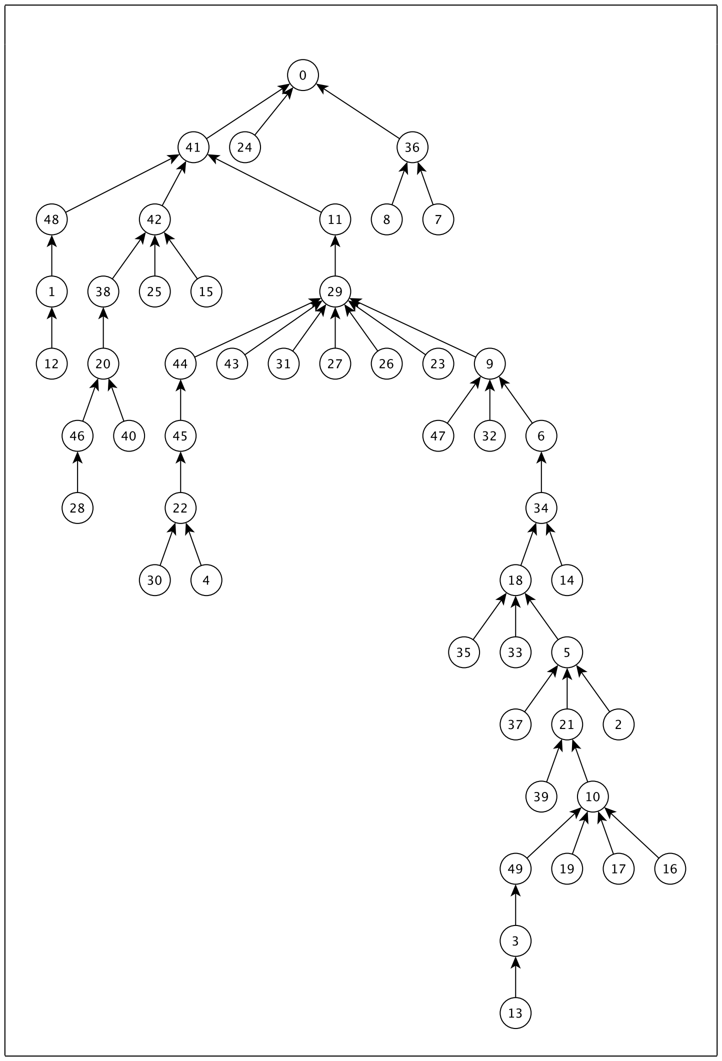

Figure 4.2. TPSN Spanning Tree

Figure 4.3 shows how long it takes to build spanning tree. This part of the algorithm is not changed except dynamic balancing enhancements described in Section 3.1.1. So, this figure shows the time complexity of reference TPSN to build a spanning tree.

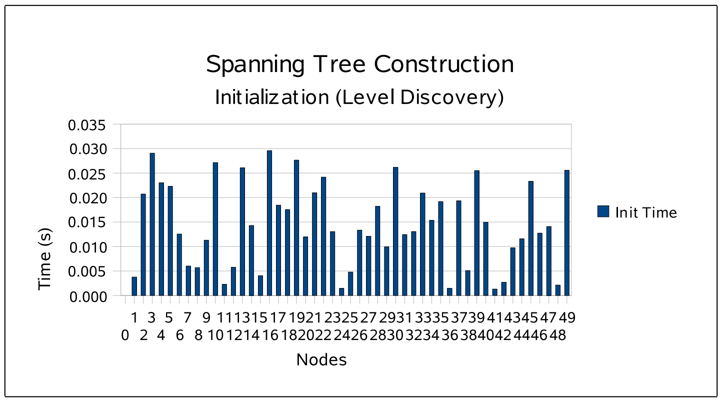

Figure 4.3. Spanning Tree Construction Time With Level Discovery

At most, 30ms is required for a node to be initialized in the spanning tree. That is really a tiny value. Also, as shown in the Figure 4.2, it takes longer for higher level nodes to be initialized as expected. Because, they wait their parent nodes to set their levels and sent level discovery packet. After getting a level discovery packet, each node waits around 10us to avoid collisions. So, it is possible to say that, it takes 30ms to build a TPSN spanning tree.

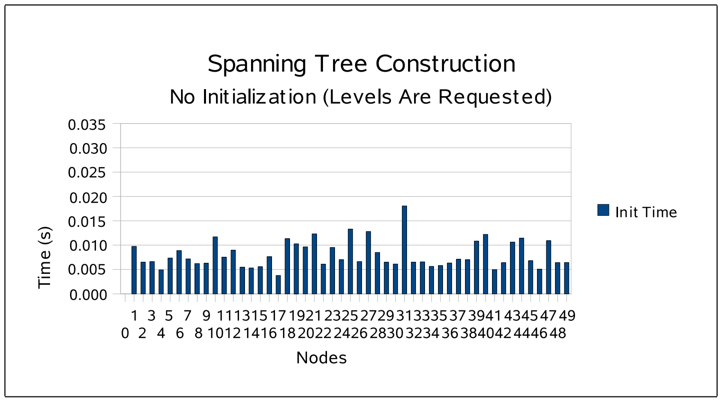

Figure 4.4. Spanning Tree Construction Time(Init) With Level Request

Another time complexity measurement is measuring the time of adding a new node to the network (Bettstetter and König 2002). To simulate this, only the root node is set and level initialization phase is skipped. That means each node will now send level request packet. Figure 4.4 shows the initialization time for each node requesting level.

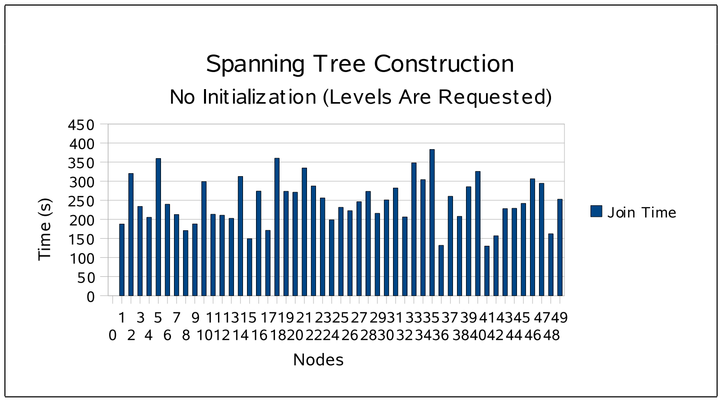

Figure 4.5. Spanning Tree Construction Time(Join) With Level Request

It is clear that level requesting takes less time because it waits just a reply packet from one-hop neighbour. So, it requires less time to initialize new level. However, to start a level request, implementation should wait for a while to be sure, which makes level request initialization very longer. This wait time is included in the Figure 4.5 which nodes use 100s wait time and adaptive synchronization interval enhancement is enabled, which is described in Section 3.3. The reason seeing join times up to 400s is, each node waits their parent to join the network. Therefore, more than one request might be required to join the network.

After spanning tree is constructed, it is now possible to synchronize the network using reference TPSN behaviour. Figure 4.6 shows the results of TPSN applied with no change using a sync interval of 100s. The improvement in synchronization can be seen easily. However, there is still a considerable clock difference. Decreasing the interval will improve the synchronization but it will also increase the message load of the system.

Figure 4.7 shows the clock difference on each node. Form now on, this type of figure will be used for the enhancement results. Because, as shown in Figure 4.6, it seems completely synchronized, which is not totally true as Figure 4.7 shows. Table B.1 is available in appendices in order to see all the gathered data for the results of reference TPSN implementation.

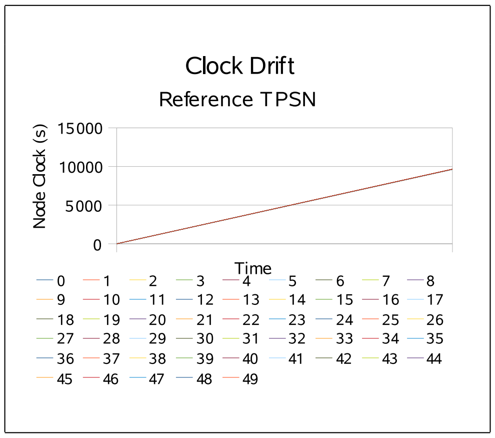

Figure 4.6. Clock Drifts Of Nodes (Reference TPSN)

In addition to time complexity, results are available after measuring message complexity of reference TPSN. Figure 4.8 shows the messages sent/received in order to synchronize the network. As seen from the figure, node 29 is highly loaded and sent/received counts are not balanced well in the network. The enhancements should get more balanced results and decrease message complexity while increasing clock precision.

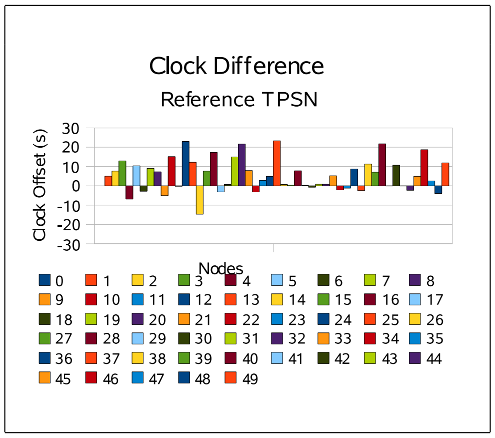

Figure 4.7. Clock Difference Of Nodes (Reference TPSN)

4.2.2. Enhancement Results

In this part, the enhancements described in Chapter 3 are applied to the simulation and gathered the results. In short, it is observed that all the enhancements made positive difference in synchronization results and provided higher precision in synchronization.

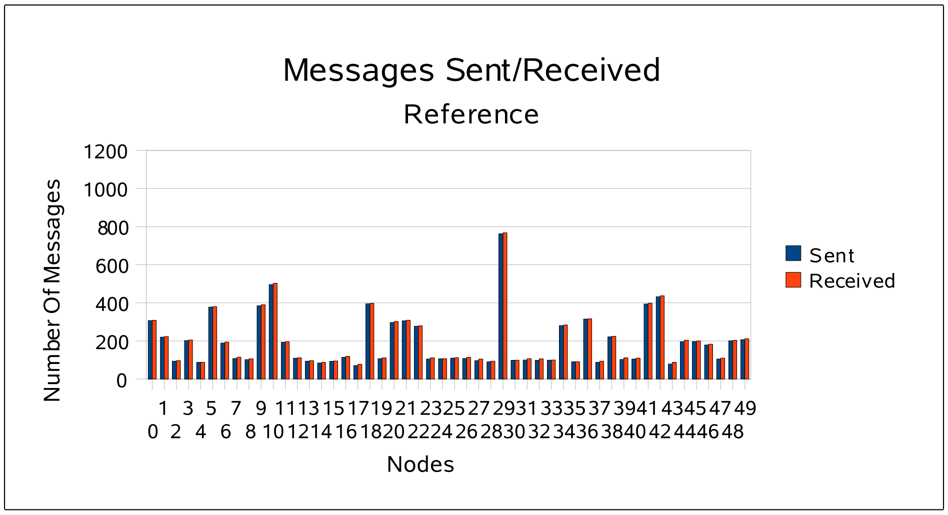

Figure 4.8. Clock Difference Of Nodes (Reference TPSN)

4.2.2.1. Clustering

Using max level of depth value as 4, clustered spanning tree as shown in Figure 3.1 becomes available. The synchronization interval is kept at 100s however, inter-cluster synchronization uses two times longer interval, which is $2 * 100$s. Figure 4.9 shows the results for clustering enhancement.

There is no change in synchronization inside a cluster. They continue to use pure TPSN synchronization method inside. However, between cluster roots, the chain synchronization method is used which is described in Section 3.2 with a longer interval (200s). By using chain synchronization between cluster roots, time precision of cluster roots is increased directly and other nodes indirectly.

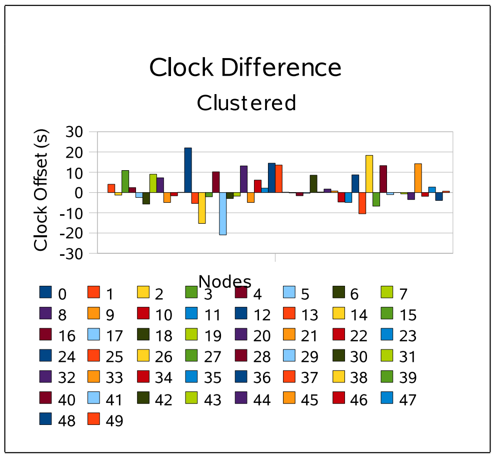

Figure 4.9. Clock Difference Of Nodes (Clustering)

As expected, synchronization between cluster roots increased the precision and decreased total clock difference. As TPSN causes bigger clock differences at higher levels, dividing the network into clusters and synchronizing the cluster heads causes cluster intermediate nodes to have a better synchronized clocks. However, as Figure 4.9 shows, there are still some problems with the higher level nodes in the clusters such as 12 and 17.

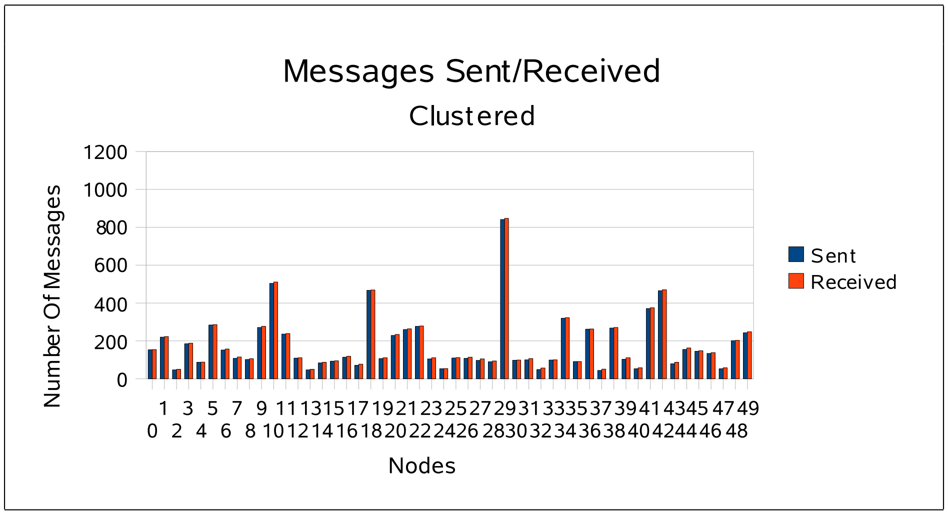

Figure 4.10. Message Complexity (Clustered TPSN)

If the messages required are analysed, it is seen that the total messages sent/received are decreased. That's very important because the precision is increased here by decreasing message count as shown in Figure 4.10. The decrease is caused by the new synchronization interval of cluster roots, which is two times longer. The improvement can easily be seen for nodes such as 0, 5 or 9 on the Figure 4.8 and 4.10.

At first, using another path among cluster roots had been preferred however, after seeing the effect of this method, it is decided to use existing tree and not to increase implementation complexity. As shown in Figures 5.1 and 5.2, the clock offset difference and message count are both decreased. This might not be the case all the time but such an improvement is expected. Because, although message count is increased to synchronize cluster roots, the interval is decreased to synchronize. Detailed results are available in appendices at Table B.2

Simulation results of dynamic balancing enhancements introduced in sections 3.1.1.1 and 3.1.1.2 will be explained in sections 4.2.2.4 and 4.2.2.5.

4.2.2.2. Chain Synchronization

In clustering enhancement, pure TPSN is kept running inside the clusters. In this enhancement, chain synchronization is used inside clusters with some break conditions as described in Section 3.2 without disabling clustering described in Section 4.2.2.1,

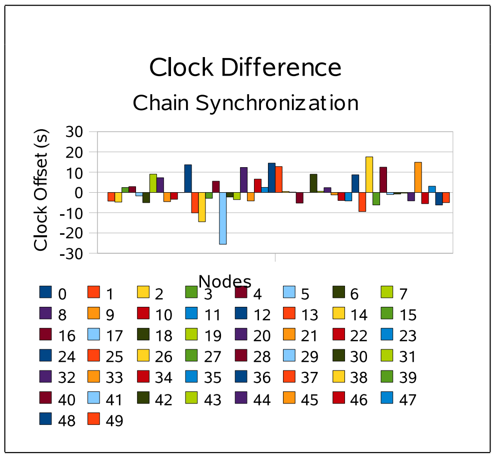

Figure 4.11. Clock Difference Of Nodes (Chained TPSN)

As Figure 4.11 shows, there is a better synchronization among nodes with a small message count increase. This message overhead again is expected because, in this enhancement, chain synchronization is done for all the nodes when available. Results for nodes, such as 12, show the effect of the enhancement.

As shown in Figures 5.1 and 5.2, less messages are used to have better synchronization however the number of messages required are increased over clustering enhancement. Figure 4.12 shows message count of each node.

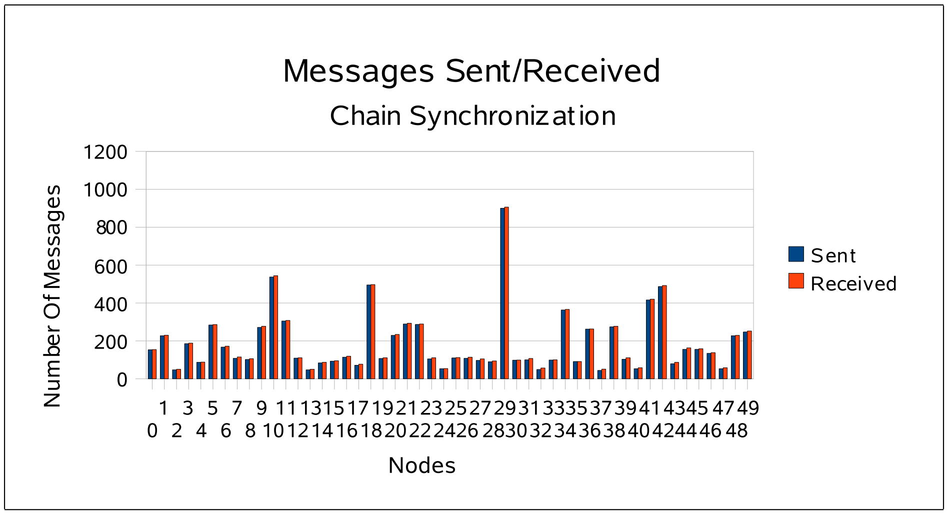

Figure 4.12. Message Complexity (Chain Synchronization)

Because of the fact that chain synchronization requires more messages, such an increase is expected in the message complexity results. Having this little increase in precision, this enhancement may not be suitable all the time but it can be used where higher precision is required with a little message count increase. As shown in Figure 4.12, the node 29 is now loaded more. These message counts should be balanced while not decreasing the precision. Table B.3 at appendices shows all the data gathered from *ns2*.

4.2.2.3. Adaptive Synchronization Interval

If the nodes had fixed clock drift, then there would not be any need for such an enhancement. However, that is not the case. Clock drifts can be changed after a while. Also, it can be very costly to set interval values for each nodes individually at the beginning. Therefore, this enhancement might be a good synchronization optimizer.

As described in Section 3.3, it changes synchronization interval in small steps (1%) until a limit value is reached. In this simulation, an interval change limit is not used but clock difference allowed maximum value is used as 1/10 of the synchronization interval. So, a 10s clock difference is expected using this enhancement.

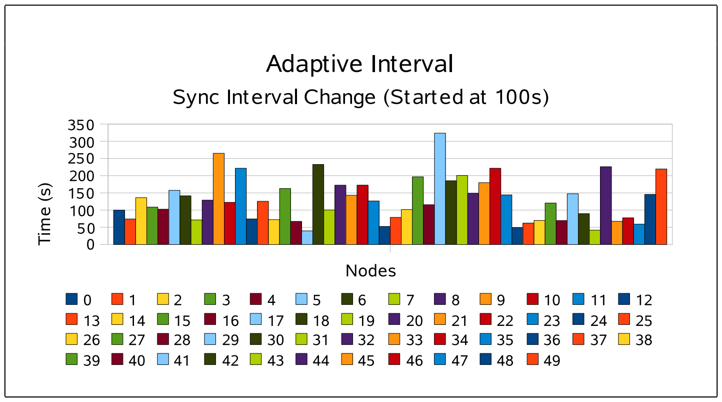

Figure 4.13. Adaptive Synchronization Interval Change

Figure 4.13 shows how synchronization intervals are changed to meet the offset requirements. For instance, it can be seen that node 9 and 29 has lower drift rates however, nodes 17 and 43 has higher drift rates.

This enhancement is here to balance message traffic to reduce message count if possible. So, the enhancement is enabled in the implementation without disabling the others and the results are shown in Figures 5.1 and 5.2.

The goal here is reducing the message traffic which is done perfectly. However, interestingly the precision is increased again. This might be possible after selecting a good expected clock difference value for adaptive synchronization. This parameter is very important to make adaptive synchronization interval work properly.

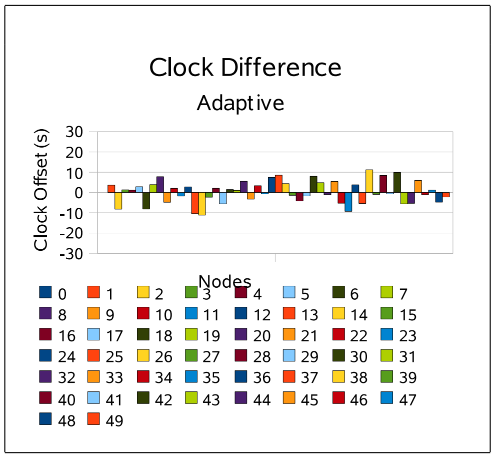

Figure 4.14. Clock Difference Of Nodes (Adaptive Synchronization Interval)

As Figure 4.14 shows, best clock difference results are achieved now and in total, minimum number of messages are used. Figure 4.15 shows number of messages sent/received for each node and the decrease can easily be seen in this figure. For the nodes the interval is increased, the messages are decreased. However, for the nodes the interval is decreased, the messages are increased as seen from the figures.

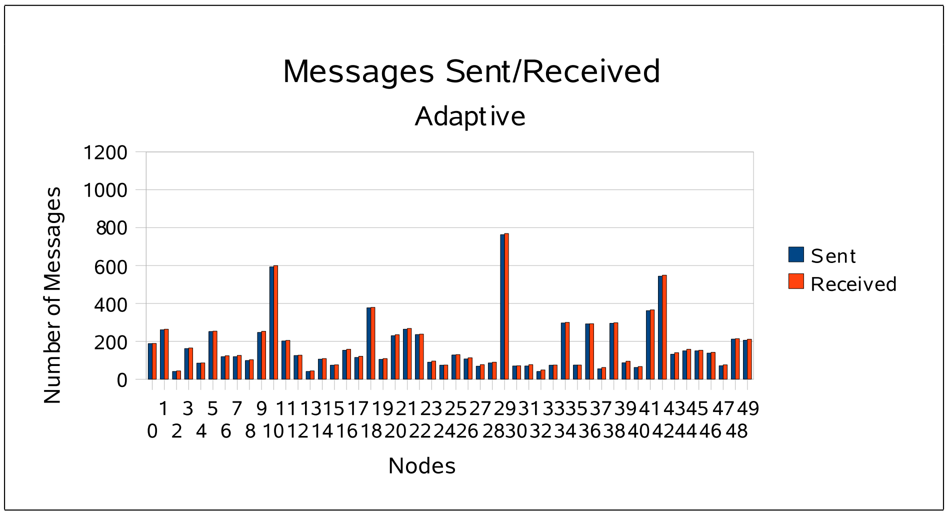

Figure 4.15. Message Complexity (Adaptive Synchronization)

Although, the results are very good here, a balanced message count for nodes is still not available. Following enhancements should fix this. Table B.4 in appendices shows the details of the results.

4.2.2.4. Balanced Children Count

Balancing is here mainly for dense network protection. The effects of this enhancement over dense networks can be seen in Section 3.1.1.1. However, to see the effects on the reference network, 4 is used as maximum child count on spanning tree shown in Figure 4.2 so this new spanning tree is constructed as shown in Figure 3.6. After applying this enhancements, node 29 has less children now.

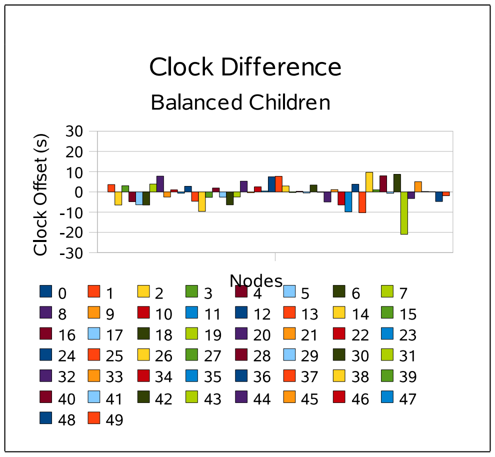

Figure 4.16. Clock Difference Of Nodes (Balanced Children)

This will reduce the synchronization request load on node 29 so it will reduce message complexity on that node. However, nodes disconnected from 29 connect to other nodes so the message load is increased on new parents. This balances message count on nodes. For instance, node 43 is disconnected from node 29 and connected to node 26. As shown in Figure 4.17, node 26 sent and received more messages than before but node 29 has less.

There is no important decrease or increase in the clock difference as shown in Figure 4.16. However, some individual nodes has higher clock difference now such as node 43. The reason must be increased level of that node.

In Figures 5.1 and 5.2, this enhancement reduces total number of messages sen-

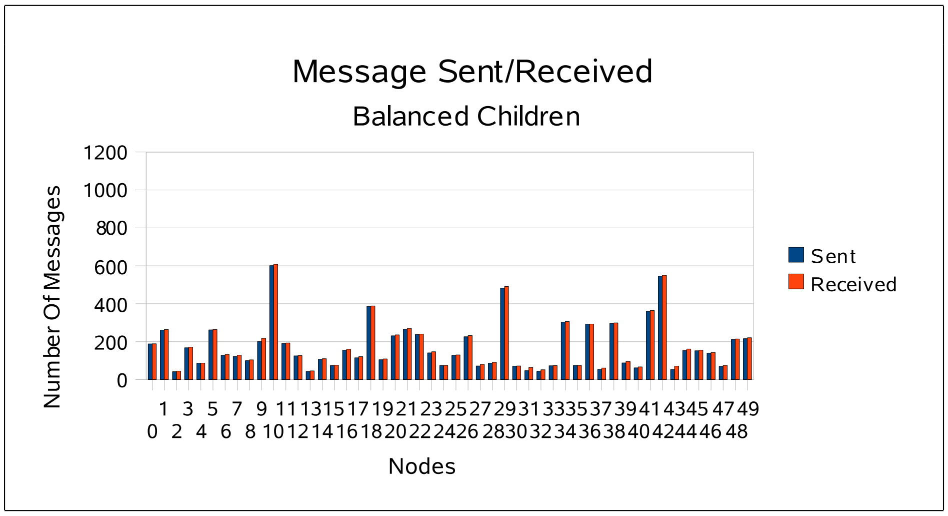

Figure 4.17. Message Complexity (Balanced Children)

t/received and the standard deviation is now less than the reference. Although the precision of clock difference is affected a bit from this situation, it is still relatively good.

4.2.2.5. Balanced Cluster Node Count

Balancing cluster node count enhancement can avoid big cluster node count differences and this should decrease the message load of the network as well. If 12 is used as maximum node count in a cluster and all other enhancements are not disabled, Figure 3.8 shows the spanning tree after using this dynamic clustering algorithm. The balancing algorithm is rather primitive however, it really improves the balance of the clustering. So, complex algorithms may not be needed to get better results all the time.

Figure 4.18 shows the clock difference results of the simulation. Average value is the worst among all enhancements (still far better than reference TPSN) however, standard deviation of clock difference is the minimum among all of other enhancements. That means, the clocks become very close to each other with this enhancement.

For message complexity, standard deviation is again minimum as clock difference. Figure 4.19 shows the results. Having minimum standard deviation in message counts means that the most balanced network for message complexity is achieved.

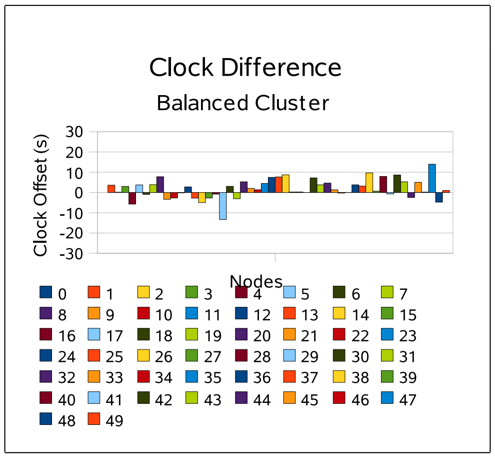

Figure 4.18. Clock Difference Of Nodes (Balanced Cluster)

As Figures 5.1 and 5.2 show, balancing cluster node count work well but results in losing some clock difference precision here. This may usually be acceptable because with such enhancements, it is possible to keep the network longer with all the nodes running.

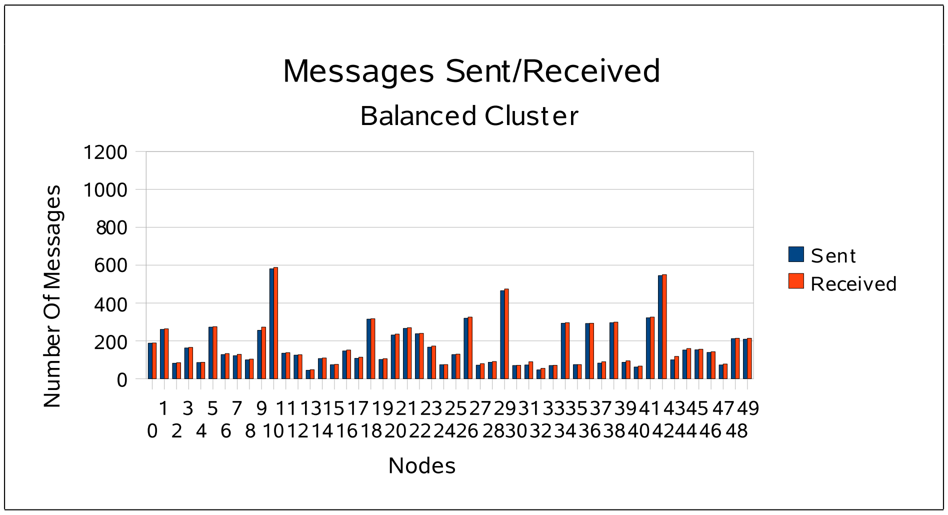

Figure 4.19. Message Complexity (Balanced Cluster)

CHAPTER 5

CONCLUSION

Although TPSN produces a good synchronization in the WSN, the improvement methods given here are providing reasonable precision improvement with low message overhead. Figures 5.1 and 5.2 shows the improvements on the synchronization of the methods given here. Individual results mean that each enhancement is enabled while all the others are disabled. Cumulative results show the results where each enhancement is enabled without disabling the previous enhancements in order.

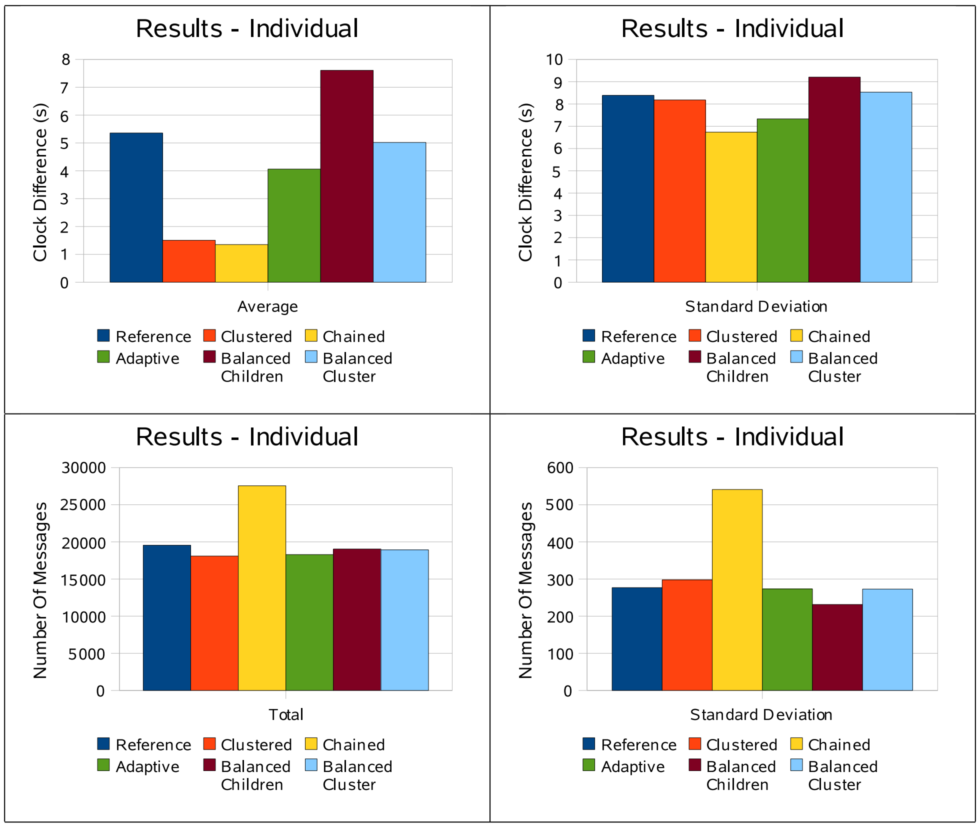

Figure 5.1. Individual Results Of The Enhancements

First improvement is making network hierarchy clustered and doing chain synchronization between cluster roots. As seen in the Figures 5.1 and 5.2, both average clock difference and the standard deviation are decreased and message traffic is not increased. Using longer intervals between cluster roots is the key here to reduce message count.

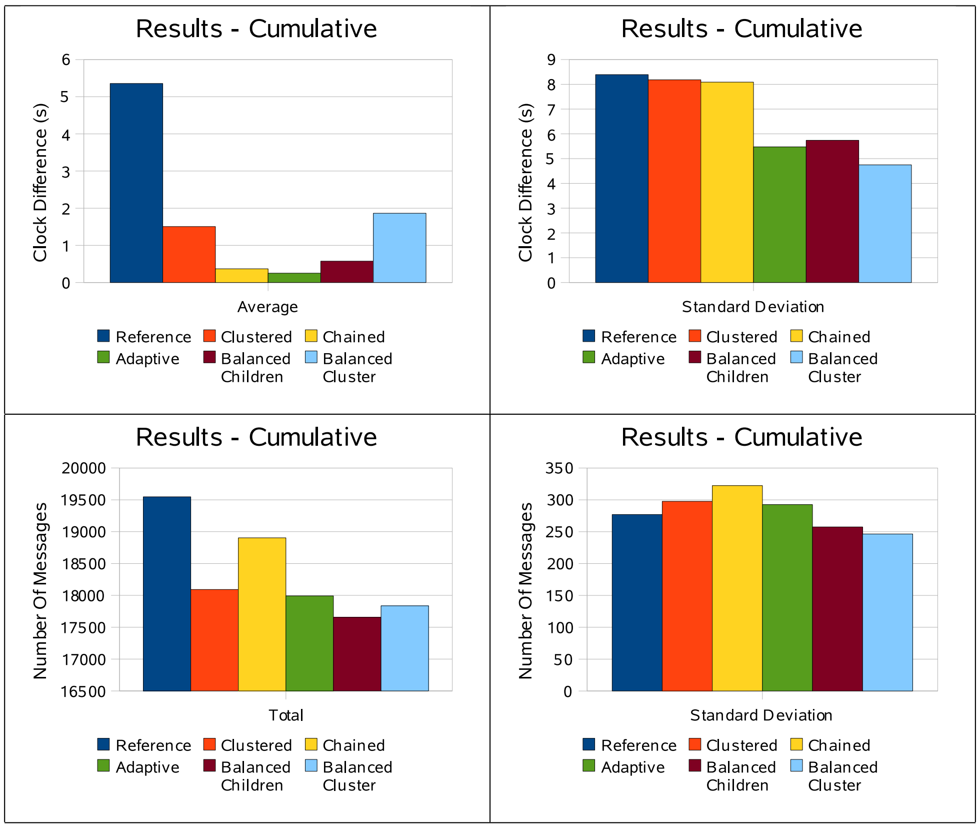

Figure 5.2. Cumulative Results Of The Enhancements

Second enhancement is applying chain synchronization to all nodes. In this method, all nodes try to chain synchronize at first. However, it might not be possible to do it all the time so not continuing the chain is more convenient. If that is the case, parent node replies to the request as similar to what reference TPSN does. Now, the precision of synchronization is improved again however there is a message overhead onto clustered TPSN. This enhancement introduces some break conditions to stop chain synchronization on any intermediate node to reduce huge message overhead of using multi-hop synchronization without interfering the precision too much.

Last improvement is a bit tidying up method to remove extra message overhead.

However, it shows that it also enhances the synchronization results. The reason behind this might be balanced message traffic. Adaptive interval offset parameter is absolutely crucial here. Any improper value given can make network traffic down or synchronization bad.

In addition to those three enhancements to improve synchronization, there are two other enhancements to make more balanced spanning tree and clusters. First enhancement limits the children of a node can have so this distributes message overhead of the nodes which have many children especially in dense networks. Although this enhancement affects clock difference a bit, it optimizes message load of the nodes. Second enhancement uses some primitive but effective conditions to join nodes to or disconnect nodes from a cluster. Applying this enhancement produces the best balanced message traffic among all but it cannot be said for the clock difference.

Adaptive synchronization seems the most effective method among all those enhancements especially used with clustering. It has good results individually as well. However, chain synchronization itself increases message count too much so practically cannot be used without clustering. Also, clustering itself makes an important difference in the precision without increasing message count. Enabling all these there enhancements at the same time gives the best result in the simulation. That means they all work properly and there is no conflict among them. On the other hand, balancing enhancements seems not mature enough and they can be improved.

In conclusion, all the methods above provides significant improvements onto TPSN except balancing enhancements. However, balancing enhancements may be crucial when the nodes are getting closer to each other.

REFERENCES

Bettstetter, C. and S. König. 2002. On the message and time complexity of a distributed mobility-adaptive clustering algorithm in wireless ad hoc networks. In *Proceedings of the 4th european wireless*. Florence, Italy.

Chao-Nong, Xu, Zhao Lei, Xu Yong-Jun, and Li Xiao-Wei. 2006. Simsync: A time synchronization simulator for sensor networks. *Acta Automatica Sinica* 32(6): 1008 – 1014.

Chen, Shujuan, Adam Dunkels, Fredrik Österlind, Thiemo Voigt, and Mikael Johansson. 2007. Time synchronization for predictable and secure data collection in wireless sensor networks. In *Proceedings of the sixth annual mediterranean ad hoc networking workshop (med-hoc-net 2007)*. Corfu, Greece.

Elson, Jeremy, Lewis Girod, and Deborah Estrin. 2002. Fine-grained network time synchronization using reference broadcasts. In *Osdi*.

Ganeriwal, Saurabh, Srdjan Capkun, Chih-Chieh Han, and Mani B. Srivastava. 2005a. Secure time synchronization service for sensor networks. In *Workshop on wireless security*, ed. Markus Jakobsson and Radha Poovendran, 97–106. ACM.

Ganeriwal, Saurabh, Deepak Ganesan, Hohyun Shim, Vlasios Tsiatsis, and Mani B. Srivastava. 2005b. Estimating clock uncertainty for efficient duty-cycling in sensor networks. In *Sensys*, ed. Jason Redi, Hari Balakrishnan, and Feng Zhao, 130–141. ACM.

Ganeriwal, Saurabh, Ram Kumar, and Mani B. Srivastava. 2003. Timing-sync protocol for sensor networks. In *Sensys*, ed. Ian F. Akyildiz, Deborah Estrin, David E. Culler, and Mani B. Srivastava, 138–149. ACM.

Hu, An-Swol and Sergio D. Servetto. 2005. On the scalability of cooperative time synchronization in pulse-connected networks. *CoRR* abs/cs/0503031. Informal publication.

Li, Qun and Daniela Rus. 2006. Global clock synchronization in sensor networks. *IEEE Trans. Computers* 55(2):214–226.

Ping, Su. 2003. Delay measurement time synchronization for wireless sensor networks. Tech. Rep., Intel Research.

Saravanos, Yanos. 2006. Energy-aware time synchronization in wireless sensor networks. *University of North Texas*, *thesis of MS*.

Simeone, O. and U. Spagnolini. 2007. Distributed time synchronization in wireless sensor networks with coupled discrete-time oscillators. *EURASIP Journal on Wireless Communications and Networking* 2007:57054.

Sundararaman, Bharath, Ugo Buy, and Ajay D. Kshemkalyani. 2005. Clock synchronization for wireless sensor networks: a survey. *Ad Hoc Networks* 3(3):281–323.

Tjoa, R., K.L. Chee, P.K. Sivaprasad, S.V. Rao, and J.G. Lim. 2004. Clock drift reduction for relative time slot tdma-based sensor networks. *Personal, Indoor and Mobile Radio Communications, 2004. PIMRC 2004. 15th IEEE International Symposium on* 2:1042– 1047.

APPENDIX A

SIMULATION USING POSIX THREADS (SOURCE CODE)

main.c

```
#include <pthread.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "message_interface.h"

//#define DYNAMIC_SYNC_INTERVAL

//#define TRACE_CLOCK_CHANGE
#ifdef TRACE_CLOCK_CHANGE
#define TRACE_CLOCK_CHANGE_INTERVAL 10000
#endif

#define TRACE_LEVEL_DISCOVERY_PHASE

#define NUMBER_OF_NODES 10
#define MAX_DEPTH NUMBER_OF_NODES
#define LEVEL_REQUEST_TIMEOUT (NUMBER_OF_NODES * 200)
#define LEVEL_ANSWER_TIMEOUT 1000
#define PACKET_PREPARATION_DELAY 5

unsigned int clocks[NUMBER_OF_NODES];

#define WRAP_POINT ((unsigned int) -1)

typedef enum
{
    INITIALISING,
    REQUESTING_LEVEL,
    READY,
    SYNCING,
    END
} node_state_type;

int estimatedSourceTimeOffset(unsigned int Ljt4, unsigned int Lit2, unsigned int Lit7, unsigned int Ljt6)
{
        unsigned int offset = 0;

        while(Ljt6 < Ljt4 || Lit7 < Lit2)
        {
                if(Ljt6 < Ljt4)
                {
                        offset += WRAP_POINT - Ljt4;
                }
                else
                {
                        offset += WRAP_POINT - Lit2;
                }

                Ljt4 = (offset + Ljt4) % WRAP_POINT;
                Lit2 = (offset + Lit2) % WRAP_POINT;
                Lit7 = (offset + Lit7) % WRAP_POINT;
                Ljt6 = (offset + Ljt6) % WRAP_POINT;

        }

        return (((int)(Ljt4 - Lit2) - (int)(Lit7 - Ljt6))) / 2;
}

void *clockTicker(void *param)
{
        unsigned int ticks = 0;
        unsigned int drift[NUMBER_OF_NODES];

        int i;
        for (i = 0; i < NUMBER_OF_NODES; ++i)
        {
                clocks[i] = 0;
                drift[i] = 5 + (rand() % 10);
        }

        for (;;)
        {
                ticks++;
                for (i = 0; i < NUMBER_OF_NODES; ++i)
                {
                        clocks[i] = (clocks[i] + 1) % WRAP_POINT;
                        if(0 == ticks % drift[i])
                        {
                                clocks[i]++;
                        }
                }

                #ifdef TRACE_CLOCK_CHANGE
                if(0 == ticks % TRACE_CLOCK_CHANGE_INTERVAL)
                {
                        for (i = 0; i < NUMBER_OF_NODES - 1; ++i)
                        {
                                printf("%u,",clocks[i]);
                        }
                        printf("%u\n", clocks[NUMBER_OF_NODES - 1]);
                }
                #endif

        my_sleep(1);
        }
        return 0;
}

void *nodeFunction(void *param)
{
        int node_id = (int) param;
    int my_parent;
        int my_level;
        int my_state;

    if(0 == node_id) // Root node
    {
            my_msg_data_t msg_data;

                my_parent = 0;
            my_level = 0;
            my_state = READY;

                // Value has our level which is 0
                msg_data.value = my_level;
                #ifdef TRACE_LEVEL_DISCOVERY_PHASE
                printf("%d: Sending LEVEL_DISCOVERY_PACKET with level %d\n", node_id, my_level);
                #endif
            Send_Message(0, -1, msg_data, LEVEL_DISCOVERY_PACKET);
    }
    else
    {
            // -1 means uninitialised
            my_parent = -1;
            my_level = -1;
            my_state = INITIALISING;
    }

        // Level Request Timeout
        int level_request_time = 0;
    int next_sync_time = 1000;
        int msg_timeout = 50;
    int interval = 10 * msg_timeout;

    for(;;)
    {
        my_msg_t msg;

        msg = Receive_Message_Timeout(node_id, msg_timeout);

                if(INITIALISING == my_state || REQUESTING_LEVEL == my_state)
                {
                    if(0 < my_level)
                    {
                                if(LEVEL_ANSWER_TIMEOUT < clocks[node_id] - level_request_time)
                                {
                                    // Waited enough. Set state to Ready. Ignore other LEVEL_ANSWER_PACKETs
                                    #ifdef TRACE_LEVEL_DISCOVERY_PHASE
                            printf("%d: Time:%d, Requested and setting level to %d - parent %d\n", node_id, clocks[node_id], my_level, my_parent);
                            #endif
                                    my_state = READY;
                            }
                    }
                        else
                        {
                                if(LEVEL_REQUEST_TIMEOUT < clocks[node_id] - level_request_time)
                                {
                                        // No level has been set. Request level
                                my_msg_data_t msg_data;

                                level_request_time = clocks[node_id];

                                        #ifdef TRACE_LEVEL_DISCOVERY_PHASE
                                        printf("%d: No level has been set. Sending LEVEL_REQUEST_PACKET\n", node_id);
                                        #endif
                            my_state = REQUESTING_LEVEL;

                            Send_Message(node_id, -1, msg_data, LEVEL_REQUEST_PACKET);
                                }
                        }
                }
        else if(clocks[node_id] > next_sync_time)
        {
                int start_time = ((node_id) * interval) % WRAP_POINT;

                if(clocks[node_id] > start_time)
                {
                        if(0 != node_id)
                        {
                                // Every node except root will synchronise itself with its parent periodically
                                if(SYNCING == my_state)
                                {
                                        // Timeouted. Reset
                                        my_state = READY;
                                }

                                if(READY == my_state)
                                {
                                        my_msg_data_t msg_data;

                                        my_state = SYNCING;
                                        msg_data.value = 0;
                                        msg_data.s_sent_ts = clocks[node_id];
                                        Send_Message(node_id, my_parent, msg_data, SYNC_PULSE_PACKET);

                                        next_sync_time = clocks[node_id] + interval;
                                        while(WRAP_POINT < next_sync_time)
                                        {
                                                next_sync_time -= WRAP_POINT;
                                        }
                                }
                        }
                }
        }

        switch (msg.msg_type)
        {
            case LEVEL_DISCOVERY_PACKET:
                    if(INITIALISING == my_state)
                    {
                        int level = msg.data.value + 1;

                        if(0 < level && 0 > my_level)
                        {
                                my_msg_data_t msg_data;

                                #ifdef TRACE_LEVEL_DISCOVERY_PHASE
                            printf("%d: Time:%d, Setting level to %d - parent %d\n", node_id, clocks[node_id], level, msg.source);
                            #endif

                            my_level = level;
                            my_parent = msg.source;
                                            my_state = READY;

                            my_sleep(1000 * (rand() % NUMBER_OF_NODES));

                            msg_data.value = my_level;
                            if(my_level < MAX_DEPTH)
                            {
                                    #ifdef TRACE_LEVEL_DISCOVERY_PHASE
                                                    printf("%d: Sending LEVEL_DISCOVERY_PACKET with level %d\n", node_id, my_level);
                                                    #endif

                                    Send_Message(node_id, -1, msg_data, LEVEL_DISCOVERY_PACKET);
                            }
                        }
                    }
                break;
                        case SYNC_ACK_PACKET:
                    if(my_state == SYNCING)
                    {
                                        unsigned int clock = clocks[node_id];
                                        int offset = estimatedSourceTimeOffset(msg.data.r_received_ts, msg.data.s_sent_ts, clock, msg.data.r_sent_ts);
                                        unsigned int estimated_time = (clock + offset) % WRAP_POINT;

                                        #ifdef DYNAMIC_SYNC_INTERVAL
                                        int desired_avg_drift = 50;
                                        if(offset > desired_avg_drift || offset < -desired_avg_drift)
                                        {
                                                if(msg_timeout < interval)
                                                {
                                                        interval -= msg_timeout;
                                                }
                                        }
                                        else
                                        {
                                                interval += msg_timeout;
                                        }
                                        #endif

                                        if(4 == node_id)
                                        {
                                                printf("%d: Offset: %d, Interval: %d\n", node_id, offset, interval);
                                        }
                                        clocks[node_id] = estimated_time;
                                        my_state = READY;
                    }
                                break;
            case SYNC_PULSE_PACKET:
                    if(INITIALISING != my_state && REQUESTING_LEVEL != my_state)
                    {
                                msg.data.r_received_ts = clocks[node_id];

                                my_sleep(PACKET_PREPARATION_DELAY);

                            msg.data.r_sent_ts = clocks[node_id];
                            Send_Message(node_id, msg.source, msg.data, SYNC_ACK_PACKET);
                    }
                    break;
                        case LEVEL_ANSWER_PACKET:
                                if(REQUESTING_LEVEL == my_state)
                                {
                                int level = msg.data.value + 1;

                                if(0 < level && level > my_level)
                                {
                                    my_level = level;
                                    my_parent = msg.source;
                                }
                                }
                                break;
            case LEVEL_REQUEST_PACKET:
                if(INITIALISING != my_state && REQUESTING_LEVEL != my_state)
                {
                        my_msg_data_t msg_data;

                    msg_data.value = my_level;
                    Send_Message(node_id, msg.source, msg_data, LEVEL_ANSWER_PACKET);
                }
                break;
            default:
                break;
        }
    }

        return 0;
}

int main()
{
        pthread_attr_t thread_attr;
        pthread_t threads[NUMBER_OF_NODES];

    initialise_messaging(NUMBER_OF_NODES);

    pthread_attr_init(&thread_attr);
    pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_JOINABLE);

        pthread_create(&threads[0], &thread_attr, clockTicker, (void *)0);

    int i;
    for (i = 0; i < NUMBER_OF_NODES; ++i)
    {
        pthread_create(&threads[i], &thread_attr, nodeFunction, (void *)i);
    }

    for (i = 0; i < NUMBER_OF_NODES; i++)
    {
        pthread_join(threads[i] , NULL);
    }

    pthread_attr_destroy(&thread_attr);

        return 0;
}
```

APPENDIX B

SIMULATION USING NS2 (SOURCE CODE)

The ns2 simulation uses 4 files. ns2 configurations are in *config.tcl*, simulation main file is *tpsn.tcl* and protocol implementation is done in C and located in *tpsn_udp.h* and *tpsn_udp.cc*.

config.tcl

```
# CONFIG ##########

set opt(chan)                   Channel/WirelessChannel
set opt(prop)                   Propagation/TwoRayGround
set opt(ll)                     LL
set opt(mac)                    Mac/802_11
set opt(ifq)                    Queue/DropTail/PriQueue
set opt(ifqlen)                 50
set opt(netif)                  Phy/WirelessPhy
set opt(rp)                     DumbAgent
set opt(ant)                    Antenna/OmniAntenna

set opt(tr)                     /dev/null
set opt(namtr)                  /dev/null

#Number
set opt(nn)                     50
set opt(x)                      1000
set opt(y)                      1000
set opt(stop)                   10000
set opt(interval)               100

set clustered                   0
set chain                       0
set adaptive                    0
set balanced_children           0
set balanced_cluster            0

if [expr $clustered > 0] {
    puts "ENABLED: Clustering"
    set opt(chain_sync)                 1
    set max_depth                       4
} else {
    puts "DISABLED: Clustering"
    set opt(chain_sync)                 1
    set max_depth                       $opt(nn)
}

if [expr $chain > 0] {
    puts "ENABLED: Chain Synchronization"
    set opt(chain_sync_all)             1
    set opt(chain_sync_interrupt)       1
} else {
    puts "DISABLED: Chain Synchronization"
    set opt(chain_sync_all)             0
    set opt(chain_sync_interrupt)       0
}

if [expr $adaptive > 0] {
    puts "ENABLED: Adaptive Synchronization"
    set opt(keep_diff)                  [expr $opt(interval) / 10]
} else {
    puts "DISABLED: Adaptive Synchronization"
    set opt(keep_diff)                  0
}

if [expr $balanced_children > 0] {
    puts "ENABLED: Balancing Children Count"
    set max_child                       4
} else {
    puts "DISABLED: Balancing Children Count"
    set max_child                        0
}

if [expr $balanced_cluster > 0] {
    puts "ENABLED: Balancing Cluster Node Count"
    set max_node_in_cluster             12
} else {
    puts "DISABLED: Balancing Cluster Node Count"
    set max_node_in_cluster             0
}

#--------------------
set opt(seed)                   0
```

tpsn.tcl

```
source config.tcl

set ns_ [new Simulator on]
set chan [new $opt(chan)]
set prop [new $opt(prop)]

set tracefd [open $opt(tr) w]
$ns_ trace-all $tracefd
set namtracefd [open $opt(namtr) w]
$ns_ namtrace-all-wireless $namtracefd $opt(x) $opt(y)

set topo [new Topography]
$topo load_flatgrid $opt(x) $opt(y)

create-god [expr $opt(nn)]

#Define a 'finish' procedure
proc finish {} {
        global ns_ tracefd namtracefd opt
        $ns_ flush-trace

        close $namtracefd
        close $tracefd

        #exec nam $opt(namtr) &
        exit 0
}

$ns_ node-config -adhocRouting $opt(rp) \
                 -llType $opt(ll) \
                 -macType $opt(mac) \
                 -ifqType $opt(ifq) \
                 -ifqLen $opt(ifqlen) \
                 -antType $opt(ant) \
                 -propInstance $prop \
                 -phyType $opt(netif) \
                 -channel $chan \
                 -topoInstance $topo \
                 -agentTrace ON \
                 -routerTrace OFF \
                 -macTrace ON \
                 -movementTrace OFF

# Reset seed
expr srand(0)

for {set i 0} {$i < $opt(nn) } {incr i} {
    set node_($i) [$ns_ node]

    set dest_x [expr rand() * $opt(x)]
    set dest_y [expr rand() * $opt(y)]

    # Put node randomdy in a place
    $node_($i) set X_ $dest_x
    $node_($i) set Y_ $dest_y
    $node_($i) set Z_ 0

    # No move
    $node_($i) random-motion 0
    $ns_ at 0 "$node_($i) setdest $dest_x $dest_y 0"

    set udp_($i) [new Agent/TPSNUDP]
    $ns_ attach-agent $node_($i) $udp_($i)
    $udp_($i) set state_ 0
    $udp_($i) set interval_ $opt(interval)
    $udp_($i) set keep_clock_difference_around_ $opt(keep_diff)
    $udp_($i) set chain_sync_enabled_ $opt(chain_sync)
    $udp_($i) set chain_sync_enabled_for_all_ $opt(chain_sync_all)
    $udp_($i) set chain_sync_interrupt_enabled_ $opt(chain_sync_interrupt)
}

for {set i 0} {$i < [expr $opt(nn) - 1] } {incr i} {
    for {set j [expr $i + 1]} {$j < $opt(nn) } {incr j} {
        $ns_ connect $udp_($i) $udp_($j)
        #puts "Connecting $i to $j"
        #$ns_ at [expr 100 * ($i + 1)] "$udp_($i) sync"
    }
}

for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at [expr $opt(interval) * rand()] "$udp_($i) initialise"
}

for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at [expr $opt(interval) * rand()] "$udp_($i) enable_child_sync_timer"
}

for {set i 0} {$i < $opt(nn)} {incr i} {
    $ns_ at 0 "$udp_($i) setasalive"
    $ns_ at 0 "$udp_($i) setmaxdepth$max_depth"
    $ns_ at 0 "$udp_($i) setmaxchild$max_child"
    $ns_ at 0 "$udp_($i) setmaxnodeincluster$max_node_in_cluster"
}

$ns_ at 0 "$udp_(0) setasroot"

for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at [expr $opt(stop) - $opt(interval)] "$udp_($i) local_clock"
}

for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at [expr $opt(stop) - $opt(interval)] "$udp_($i) time_complexity"
}

#Call the finish procedure
$ns_ at $opt(stop) "finish"

#Run the simulation
puts "Starting Simulation..."
$ns_ run
```

tpsn_udp.h

```
#ifndef NS_TPSN_UDP_H
#define NS_TPSN_UDP_H

#include "udp.h"

enum messageType
{
        INVALID_PACKET,
        LEVEL_DISCOVERY_PACKET,
        LEVEL_REQUEST_PACKET,
        LEVEL_ANSWER_PACKET,
        SYNC_PULSE_PACKET,
        SYNC_GROUP_PULSE_PACKET,
        SYNC_ACK_PACKET,
        SYNC_GROUP_ACK_PACKET,
        SYNC_REQUEST_PACKET,
        SYNC_ACK_DISCONNECT_PACKET,
        MESSAGE_TYPE_SIZE, // Must be at the and
};

enum nodeState
{
        INITIALISING,
        REQUESTING_LEVEL,
        READY,
        SYNCING,
        CHAIN_SYNCING
};

struct ChildNode
{
        int node_id;
        bool alive;
        int descendant_count;
        ChildNode(): node_id(0), alive(false)
        {

        }
};

class TpsnUdpAgent;
class TpsnSyncTimer : public TimerHandler
{
public:
        TpsnSyncTimer(TpsnUdpAgent *agent);
        TpsnSyncTimer();
        virtual ~TpsnSyncTimer();
        void setAgent(TpsnUdpAgent *agent);

protected:
        virtual void expire(Event *event);
        TpsnUdpAgent *m_agent;
private:
};

class TpsnUdpAgent : public UdpAgent
{
public:
        TpsnUdpAgent();
        TpsnUdpAgent(packet_t);
        virtual void sendmsg(int nbytes, AppData* data, nsaddr_t daddr, const char *flags = 0);
        virtual void recv(Packet* pckt, Handler* hndlr);
        virtual int command(int argc, const char*const* argv);

        void timeout(int);
protected:
        static double estimatedSourceTimeOffset(double Ljt4, double Lit2,
                        double Lit7, double Ljt6);
        double localClockCurrent();
        double localClock(double global_clock);
        void setOffset(double offset);
        bool isMyChild(int child_node_id);
        bool setAsMyChild(int child_node_id, int descendant_count = 0);
        bool updateMyChild(int child_node_id, int descendant_count = 0);
        bool removeFromMyChildren(int child_node_id);
        int descendantCount();
        bool disconnectNodeToFormNewCluster(int child_node_id);
        bool connectNodeToCluster(int child_node_id);

        nodeState m_state;
        int m_level;
        int m_group;
        nsaddr_t m_parent;
        nsaddr_t m_child_for_chain_synching;
        double m_child_sent_ts_for_chain_synching;
        double m_clock_offset;
        double m_clock_drift_multiplier;
        TpsnSyncTimer m_tpsn_sync_timer;
        double m_interval;
        double m_keep_clock_difference_around;
        bool m_child_sync_timer;
        int m_chain_sync_interval_count;
        double m_last_sync_time;
        int m_chain_sync_enabled;
        int m_chain_sync_enabled_for_all;
        int m_chain_sync_interrupt_enabled;
        int m_received_message_count[MESSAGE_TYPE_SIZE];
        int m_sent_message_count[MESSAGE_TYPE_SIZE];
        int m_timeout_count;
        int m_sync_fail_count;
        double m_init_time;
        double m_inited_time;
        bool m_alive;
        int m_max_depth;
        int m_max_child;
        int m_max_node_in_cluster;
        int m_child_count;
        ChildNode *m_child_list;
        int m_root_wait_count;
        int m_level_request_count;
};

struct TpsnData
{
        TpsnData() :
                type(INVALID_PACKET), level(-1), group(-1), sender(-1), s_sent_ts(-1), r_received_ts(-1),
                                r_sent_ts(-1), start_new_cluster(false), join_to_cluster(false)
        {
        }

        messageType type;
        int level;
        int group;
        int sender;
        double s_sent_ts;
        double r_received_ts;
        double r_sent_ts;
        int descendant_count;
        bool start_new_cluster;
        bool join_to_cluster;
};
#endif
```

tpsn_udp.cc

```
#undef NDEBUG
#include "tpsn_udp.h"
#include "rtp.h"
#include "random.h"
#include "address.h"
#include "ip.h"
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
double rand_drift_multipier()
{
        return (Random::normal(1, 0.1));
}

void TpsnUdpAgent::setOffset(double offset)
{
        m_clock_offset += offset;
        m_last_sync_time = localClockCurrent();
        if (m_keep_clock_difference_around > 0)
        {
                double ratio = m_keep_clock_difference_around / offset;
                if (ratio < 0)
                {
                        ratio = 0 - ratio;
                }

                if (ratio < 1)
                {
                        ratio = 0.99; // 1% decrement
                }
                else if (ratio > 1)
                {
                        ratio = 1.01; // 1% increment
                }
                else
                {
                        ratio = 1;
                }

                //printf("%d: Trying to keep clock offset around %lf, Current: %lf\n", (int)addr(), m_keep_clock_difference_around, offset);
                //printf("%d: Ratio: %lf\n", (int)addr(), ratio);
                m_interval *= ratio;

                //printf("%d: New interval: %lf\n", (int)addr(), m_interval);
        }
}

static class TpsnUdpAgentClass: public TclClass
{
public:
        TpsnUdpAgentClass() :
                TclClass("Agent/TPSNUDP")
        {
        }
        TclObject* create(int, const char*const*)
        {
                return (new TpsnUdpAgent());
        }
} class_tpsn_udp_agent;

TpsnUdpAgent::TpsnUdpAgent() :
        m_state(INITIALISING), m_level(-1), m_group(-1), m_parent(0),
                        m_child_for_chain_synching(-1), m_child_sent_ts_for_chain_synching(
                                        -1), m_clock_offset(0), m_clock_drift_multiplier(
                                        rand_drift_multipier()), m_interval(0),
                        m_keep_clock_difference_around(0), m_child_sync_timer(false),
                        m_chain_sync_interval_count(0), m_last_sync_time(
                                        localClockCurrent()), m_chain_sync_enabled(0),
                        m_chain_sync_enabled_for_all(0), m_chain_sync_interrupt_enabled(0),
                        m_timeout_count(0), m_sync_fail_count(0), m_init_time(0),
                        m_inited_time(0), m_alive(false), m_max_depth(0), m_max_child(0),
                        m_max_node_in_cluster(0), m_child_count(0), m_child_list(0), m_root_wait_count(0), m_level_request_count(0)
{
        bind("interval_", &m_interval);
        bind("keep_clock_difference_around_", &m_keep_clock_difference_around);
        bind("chain_sync_enabled_", &m_chain_sync_enabled);
        bind("chain_sync_enabled_for_all_", &m_chain_sync_enabled_for_all);
        bind("chain_sync_interrupt_enabled_", &m_chain_sync_interrupt_enabled);

        for (int i = 0; i < MESSAGE_TYPE_SIZE; ++i)
        {
                m_received_message_count[i] = 0;
                m_sent_message_count[i] = 0;
        }

        m_tpsn_sync_timer.setAgent(this);
}

TpsnUdpAgent::TpsnUdpAgent(packet_t type) :
        UdpAgent(type), m_state(INITIALISING), m_level(-1), m_group(-1),
                        m_parent(0), m_child_for_chain_synching(-1),
                        m_child_sent_ts_for_chain_synching(-1), m_clock_offset(0),
                        m_clock_drift_multiplier(rand_drift_multipier()), m_interval(0),
                        m_keep_clock_difference_around(0), m_child_sync_timer(false),
                        m_chain_sync_interval_count(0), m_last_sync_time(
                                        localClockCurrent()), m_chain_sync_enabled(0),
                        m_chain_sync_enabled_for_all(0), m_chain_sync_interrupt_enabled(0),
                        m_timeout_count(0), m_sync_fail_count(0), m_init_time(0),
                        m_inited_time(0), m_alive(false), m_max_depth(0), m_max_child(0),
                        m_max_node_in_cluster(0), m_child_count(0), m_child_list(0), m_root_wait_count(0), m_level_request_count(0)
{
        bind("interval_", &m_interval);
        bind("keep_clock_difference_around_", &m_keep_clock_difference_around);
        bind("chain_sync_enabled_", &m_chain_sync_enabled);
        bind("chain_sync_enabled_for_all_", &m_chain_sync_enabled_for_all);
        bind("chain_sync_interrupt_enabled_", &m_chain_sync_interrupt_enabled);

        for (int i = 0; i < MESSAGE_TYPE_SIZE; ++i)
        {
                m_received_message_count[i] = 0;
                m_sent_message_count[i] = 0;
        }

        m_tpsn_sync_timer.setAgent(this);
}

TpsnSyncTimer::TpsnSyncTimer(TpsnUdpAgent *agent) :
        TimerHandler()
{
        m_agent = agent;
}

TpsnSyncTimer::TpsnSyncTimer()
{
        m_agent = 0;
}

TpsnSyncTimer::~TpsnSyncTimer()
{

}

void TpsnSyncTimer::expire(Event *e)
{
        m_agent->timeout(0);
}

void TpsnSyncTimer::setAgent(TpsnUdpAgent *agent)
{
        m_agent = agent;
}

void TpsnUdpAgent::timeout(int x)
{
        if (m_alive)
        {
                bool process = true;

                if (m_level == 0)
                {
                        if(m_group == 0)
                        {
                                process = false;
                        }
                        else if(m_root_wait_count++ % 2 == 0)
                        {
                                // Inter-Cluster Synchronization Is Low Priority
                                process = false;
                        }
                }

                if(process)
                {
                        //printf("%02d: Timeout\n", (int)addr());
                        TpsnData tpsn_data;
                        PacketData* data;

                        ++m_timeout_count;
                        switch (m_state)
                        {
                        case SYNCING:
                        case CHAIN_SYNCING:
                                // printf("%02d: Syncing Failed\n", (int)addr());
                                // Request new level
                                // Fall through
                                ++m_sync_fail_count;
                        case INITIALISING:
                                m_state = REQUESTING_LEVEL;
                                // Fall through
                        case REQUESTING_LEVEL:
                                ++m_level_request_count;
                                m_init_time = 0 - localClockCurrent();
                                //printf("%02d: Requesting level, time %lf\n", (int) addr(), localClockCurrent());
                                tpsn_data.type = LEVEL_REQUEST_PACKET;
                                m_level = -1;
                                m_group = -1;
                                data = new PacketData(sizeof(TpsnData));
                                memcpy((char*) data->data(), &tpsn_data, sizeof(TpsnData));
                                sendmsg(sizeof(TpsnData), data, IP_BROADCAST);
                                break;
                        case READY:
                                if (m_child_sync_timer)
                                {
                                        //printf("%02d: Sending SYNC_PULSE_PACKET\n", (int)addr());
                                        tpsn_data.level = m_level;
                                        tpsn_data.group = m_group;
                                        tpsn_data.sender = (int) addr();
                                        tpsn_data.s_sent_ts = localClockCurrent();
                                        tpsn_data.descendant_count = descendantCount();

                                        if (m_chain_sync_enabled)
                                        {
                                                if (m_level == 0)
                                                {
                                                        //Always do chain sync for inter-cluster
                                                        //tpsn_data.to_root = true;
                                                        m_state = CHAIN_SYNCING;
                                                        tpsn_data.type = SYNC_GROUP_PULSE_PACKET;
                                                }
                                                else if (m_chain_sync_enabled_for_all)
                                                {
                                                        if (m_chain_sync_interrupt_enabled)
                                                        {
                                                                if (m_chain_sync_interval_count++ % m_level
                                                                                == 0)
                                                                {
                                                                        m_state = SYNCING;
                                                                        tpsn_data.type = SYNC_PULSE_PACKET;
                                                                }
                                                                else
                                                                {
                                                                        m_state = CHAIN_SYNCING;
                                                                        tpsn_data.type = SYNC_GROUP_PULSE_PACKET;
                                                                }
                                                        }
                                                        else
                                                        {
                                                                m_state = CHAIN_SYNCING;
                                                                tpsn_data.type = SYNC_GROUP_PULSE_PACKET;
                                                        }
                                                }
                                                else
                                                {
                                                        m_state = SYNCING;
                                                        tpsn_data.type = SYNC_PULSE_PACKET;
                                                }
                                        }
                                        else
                                        {
                                                m_state = SYNCING;
                                                tpsn_data.type = SYNC_PULSE_PACKET;
                                        }

                                        data = new PacketData(sizeof(TpsnData));
                                        memcpy((char*) data->data(), &tpsn_data, sizeof(TpsnData));
                                        sendmsg(sizeof(TpsnData), data, m_parent);
                                }
                                else
                                {
                                        //nothing to do
                                }
                                break;
                        default:
                                break;
                        }
                }
        }
        m_tpsn_sync_timer.resched(m_interval);
}

void TpsnUdpAgent::recv(Packet* pckt, Handler* hndlr)
{
        if (m_alive)
        {
                TpsnData *tpsn_data = (TpsnData *) pckt->accessdata();

                ++m_received_message_count[tpsn_data->type];

                /*
                 printf("Tpsn node received message\n");
                 printf("Type: %d\n", tpsn_data->type);
                 printf("level: %d\n", tpsn_data->level);
                 printf("Sender sent ts: %lf\n", tpsn_data->s_sent_ts);
                 printf("Receiver received ts: %lf\n", tpsn_data->r_received_ts);
                 printf("Receiver sent ts: %lf\n", tpsn_data->r_sent_ts);
                 */

                //if(m_state == READY)
                {
                        if (tpsn_data->type == SYNC_GROUP_PULSE_PACKET || tpsn_data->type
                                        == SYNC_PULSE_PACKET)
                        {
                                if (m_max_child > 0)
                                {
                                        bool reply = false;

                                        hdr_ip* ih = hdr_ip::access(pckt);
                                        if (!isMyChild(ih->saddr()))
                                        {
                                                reply = setAsMyChild(ih->saddr(),
                                                                tpsn_data->descendant_count);
                                        }
                                        else
                                        {
                                                updateMyChild(ih->saddr(), tpsn_data->descendant_count);
                                                reply = true;
                                        }

                                        if (!reply)
                                        {
                                                tpsn_data->r_received_ts = localClockCurrent();

                                                hdr_ip* ih = hdr_ip::access(pckt);

                                                tpsn_data->type = SYNC_ACK_DISCONNECT_PACKET;
                                                PacketData* data = new PacketData(sizeof(TpsnData));

                                                hdr_cmn* packet = hdr_cmn::access(pckt);
                                                tpsn_data->r_sent_ts = localClock(NOW + packet->txtime());

                                                memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                                                sendmsg(sizeof(TpsnData), data, ih->saddr());

                                                Packet::free(pckt);

                                                return;
                                        }

                                        if (m_max_node_in_cluster > 0)
                                        {
                                                if (tpsn_data->sender == ih->saddr() && (int) addr() != 0)
                                                {
                                                        if (m_group == tpsn_data->group)
                                                        {
                                                                tpsn_data->start_new_cluster
                                                                                = disconnectNodeToFormNewCluster(
                                                                                                ih->saddr());
                                                        }
                                                        else
                                                        {
                                                                if (m_level < m_max_depth - 1)
                                                                {
                                                                        tpsn_data->join_to_cluster
                                                                                        = connectNodeToCluster(ih->saddr());
                                                                }
                                                        }
                                                }
                                        }
                                }
                        }
                        else if (tpsn_data->type == SYNC_GROUP_ACK_PACKET || tpsn_data->type
                                        == SYNC_ACK_PACKET)
                        {

                                if (m_max_node_in_cluster > 0)
                                {
                                        if (tpsn_data->start_new_cluster)
                                        {
                                                //printf("Disconnecting\n");
                                                //printf("%02d: Setting level (%d->%d), group (%d->%d)\n", (int)addr(), m_level, 0, m_group, tpsn_data->grou
                                                m_level = 0;
                                                m_group = tpsn_data->group + 1;
                                                tpsn_data->start_new_cluster = false;
                                        }

                                        if (tpsn_data->join_to_cluster)
                                        {
                                                //printf("Joining\n");
                                                //printf("%02d: Setting level (%d->%d), group (%d->%d)\n", (int)addr(), m_level, tpsn_data->level + 1, m_grou
                                                m_level = tpsn_data->level + 1;
                                                m_group = tpsn_data->group;
                                                tpsn_data->join_to_cluster = false;
                                        }

                                        if (m_group == tpsn_data->group)
                                        {
                                                if (m_level != tpsn_data->level + 1)
                                                {
                                                        if (tpsn_data->level + 1 < m_max_depth && m_level != 0)
                                                        {
                                                                m_level = tpsn_data->level + 1;
                                                                m_group = tpsn_data->group;
                                                        }
                                                        else
                                                        {
                                                                m_level = 0;
                                                                m_group = tpsn_data->group + 1;
                                                        }
                                                }
                                        }
```

```
                        else
                        {
                                if (m_group < tpsn_data->group)
                                {
                                        m_level = tpsn_data->level + 1;
                                        m_group = tpsn_data->group;
                                }
                                else
                                {
                                        if (m_level != 0)
                                        {
                                                m_level = 0;
                                                m_group = tpsn_data->group + 1;
                                        }
                                }
                        }
                }
        }
}

/*
 printf("m_clock_drift_multiplier: %lf\n", m_clock_drift_multiplier);
 printf("Global clock : %lf\n", NOW);
 printf("Local clock : %lf\n", localClockCurrent());
 */
switch (tpsn_data->type)
{
case SYNC_PULSE_PACKET:
        if (INITIALISING != m_state && REQUESTING_LEVEL != m_state)
        {
                //printf("%02d: Received SYNC_PULSE_PACKET\n", (int)addr());
                tpsn_data->r_received_ts = localClockCurrent();

                hdr_ip* ih = hdr_ip::access(pckt);

                tpsn_data->type = SYNC_ACK_PACKET;
                tpsn_data->level = m_level;
                tpsn_data->group = m_group;
                PacketData* data = new PacketData(sizeof(TpsnData));

                hdr_cmn* packet = hdr_cmn::access(pckt);
                tpsn_data->r_sent_ts = localClock(NOW + packet->txtime());

                memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                sendmsg(sizeof(TpsnData), data, ih->saddr());
        }
        break;
case SYNC_GROUP_PULSE_PACKET:
        if (READY == m_state)
        {
                const int divide_interval = m_level + 1;

                if (m_level == 0 || ((localClockCurrent() - m_last_sync_time)
                                < (m_interval / divide_interval)))
                {
                        tpsn_data->r_received_ts = localClockCurrent();

                        hdr_ip* ih = hdr_ip::access(pckt);

                        tpsn_data->type = SYNC_GROUP_ACK_PACKET;
                        tpsn_data->level = m_level;
                        tpsn_data->group = m_group;
                        PacketData* data = new PacketData(sizeof(TpsnData));

                        //printf("CHAIN ACK %d->%d\n", (int)addr(), (int)ih->saddr());

                        hdr_cmn* packet = hdr_cmn::access(pckt);
                        tpsn_data->r_sent_ts = localClock(NOW + packet->txtime());

                        memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                        sendmsg(sizeof(TpsnData), data, ih->saddr());
                }
                else
                {
                        hdr_ip* ih = hdr_ip::access(pckt);

                        m_state = CHAIN_SYNCING;
                        m_child_for_chain_synching = ih->saddr();
                        m_child_sent_ts_for_chain_synching = tpsn_data->s_sent_ts;

                        tpsn_data->s_sent_ts = localClockCurrent();
                        //printf("CHAIN PULSE %d->%d\n", (int)addr(), m_parent);
                        tpsn_data->type = SYNC_GROUP_PULSE_PACKET;
                        tpsn_data->descendant_count = descendantCount();
                        tpsn_data->level = m_level;
                        tpsn_data->group = m_group;

                        PacketData* data = new PacketData(sizeof(TpsnData));

                        memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                        sendmsg(sizeof(TpsnData), data, m_parent);

                        m_tpsn_sync_timer.resched(m_interval);
                }
        }
        else
        {
                // Do not timeout. At least return current clock (TPSN like behaviour)
                tpsn_data->r_received_ts = localClockCurrent();

                hdr_ip* ih = hdr_ip::access(pckt);

                tpsn_data->type = SYNC_GROUP_ACK_PACKET;
                tpsn_data->level = m_level;
                tpsn_data->group = m_group;
                PacketData* data = new PacketData(sizeof(TpsnData));

                //printf("CHAIN ACK %d->%d\n", (int)addr(), (int)ih->saddr());

                hdr_cmn* packet = hdr_cmn::access(pckt);
                tpsn_data->r_sent_ts = localClock(NOW + packet->txtime());

                memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                sendmsg(sizeof(TpsnData), data, ih->saddr());
        }

case LEVEL_DISCOVERY_PACKET:
        //printf("%02d: Received LEVEL_DISCOVERY_PACKET\n", (int)addr());
        if (INITIALISING == m_state)
        {
                //printf("%02d: Processing LEVEL_DISCOVERY_PACKET\n", (int)addr());
                int level = tpsn_data->level + 1;

                if (0 < level)
                {
                        if(m_chain_sync_enabled) // Cluster
                        {
                                if (m_max_depth > level)
                                {
                                        m_level = level;
                                        m_group = tpsn_data->group;
                                }
                                else
                                {
                                        // Start a new cluster
                                        m_level = 0; // Be the root of new group
                                        m_group = tpsn_data->group + 1;
                                }
                        }
                        else
                        {
                                m_level = level;
                                m_group = tpsn_data->group;
                        }

                        /*
                         printf("%02d: Setting level to: %d group to: %d\n",
                         (int)addr(), m_level, m_group);
                         */

                        hdr_ip* ih = hdr_ip::access(pckt);
                        m_parent = ih->saddr();
                        m_state = READY;

                        m_init_time += localClockCurrent();
                        m_inited_time = localClockCurrent();

                        usleep(10 * (unsigned int) m_clock_drift_multiplier);

                        PacketData* data = new PacketData(sizeof(TpsnData));

                        tpsn_data->level = m_level;
                        tpsn_data->group = m_group;
                        tpsn_data->sender = (int) addr();
                        tpsn_data->type = LEVEL_DISCOVERY_PACKET;

                        memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                        sendmsg(sizeof(TpsnData), data, IP_BROADCAST);

                }
        }
        break;
case LEVEL_REQUEST_PACKET:
        if (INITIALISING != m_state && REQUESTING_LEVEL != m_state)
        {
                if (m_child_count < m_max_child || m_max_child == 0)
                {
                        TpsnData tpsn_data;
                        tpsn_data.level = m_level;
                        tpsn_data.group = m_group;
                        tpsn_data.sender = (int) addr();
                        tpsn_data.type = LEVEL_ANSWER_PACKET;

                        PacketData* data = new PacketData(sizeof(TpsnData));
                        memcpy((char*) data->data(), &tpsn_data, sizeof(TpsnData));
                        hdr_ip* ih = hdr_ip::access(pckt);
                        sendmsg(sizeof(TpsnData), data, ih->saddr());
                }
                else
                {
                        //printf("%02d: Too many children. Skipping request\n", (int)addr());
                }
        }
        //printf("LEVEL_REQUEST_PACKET\n");
        break;
case LEVEL_ANSWER_PACKET:
        if (REQUESTING_LEVEL == m_state)
        {
                int level = tpsn_data->level + 1;

                if ((m_level >= level - 1 && m_group >= tpsn_data->group)
                                || (m_level == -1 && m_group == -1))
                {
                        if (m_max_depth > level)
                        {
                                m_level = level;
                                m_group = tpsn_data->group;
                        }
                        else
                        {
                                // Start a new cluster
                                m_level = 0; // Be the root of new group
                                m_group = tpsn_data->group + 1;
                        }

                        hdr_ip* ih = hdr_ip::access(pckt);
                        m_parent = ih->saddr();
                        m_state = READY;

                        m_init_time += localClockCurrent();
                        m_inited_time = localClockCurrent();

                        /*
                         printf("%02d: Request: Setting level to: %d, group to: %d\n",
                         (int)addr(), m_level, m_group);
                         */

                }
                else
                {
                        /*
                         hdr_ip* ih = hdr_ip::access(pckt);
                         printf("%02d:INCOMPATIBLE ROOT: %d\n", (int)addr(), ih->saddr());
                         printf("m_level = %d\n", m_level);
                         printf("level = %d\n", level);
                         printf("m_group = %d\n", m_group);
                         printf("group = %d\n", tpsn_data->group);
                         */
                }
        }
        //printf("LEVEL_ANSWER_PACKET\n");
        break;
case SYNC_ACK_PACKET:
        //printf("%02d: Received SYNC_ACK_PACKET\n", (int)addr());
        if (m_state == SYNCING)
        {
                //printf("Processing SYNC_ACK_PACKET\n");
                double clock = localClockCurrent();
                double offset = estimatedSourceTimeOffset(
                                tpsn_data->r_received_ts, tpsn_data->s_sent_ts, clock,
                                tpsn_data->r_sent_ts);

                //printf("%02d: (%lf) Local time %lf\n", (int)addr(), m_clock_drift_multiplier, localClockCurrent());
                //double estimated_time = (clock + offset);
                //printf("Correcting time to %lf\n", estimated_time);
                setOffset(offset);

                //printf("%02d: [L:%02d G:%02d P:%02d] Corrected time %lf (Ref: %lf)\n", (int)addr(), m_level, m_group, (int)m_parent, localC
                m_state = READY;
        }
        break;
case SYNC_ACK_DISCONNECT_PACKET:
        if (m_state == SYNCING || m_state == CHAIN_SYNCING)
        {
                double clock = localClockCurrent();
                double offset = estimatedSourceTimeOffset(
                                tpsn_data->r_received_ts, tpsn_data->s_sent_ts, clock,
                                tpsn_data->r_sent_ts);

                setOffset(offset);

                if(m_state == CHAIN_SYNCING)
                {
                        PacketData* data = new PacketData(sizeof(TpsnData));

                        if (tpsn_data->sender != (int) addr())
                        {
                                tpsn_data->s_sent_ts = m_child_sent_ts_for_chain_synching;
```

```
                                                memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                                                sendmsg(sizeof(TpsnData), data, m_child_for_chain_synching);

                                                m_child_for_chain_synching = -1;
                                                m_child_sent_ts_for_chain_synching = -1;
                                        }
                                }

                                m_state = REQUESTING_LEVEL;
                                m_init_time = 0 - localClockCurrent();
                                TpsnData tpsn_data_request_level;
                                tpsn_data_request_level.type = LEVEL_REQUEST_PACKET;
                                PacketData *data = new PacketData(sizeof(TpsnData));
                                memcpy((char*) data->data(), &tpsn_data_request_level, sizeof(TpsnData));
                                sendmsg(sizeof(TpsnData), data, IP_BROADCAST);
                                m_tpsn_sync_timer.resched(m_interval);
                        }
                        break;
                case SYNC_GROUP_ACK_PACKET:
                        //printf("%02d: Received SYNC_GROUP_ACK_PACKET\n", (int)addr());
                        if (m_state == CHAIN_SYNCING)
                        {
                                PacketData* data = new PacketData(sizeof(TpsnData));

                                //printf("Processing SYNC_GROUP_ACK_PACKET\n");
                                double clock = localClockCurrent();
                                double offset = estimatedSourceTimeOffset(
                                                tpsn_data->r_received_ts, tpsn_data->s_sent_ts, clock,
                                                tpsn_data->r_sent_ts);

                                setOffset(offset);

                                if (tpsn_data->sender != (int) addr())
                                {
                                        tpsn_data->s_sent_ts = m_child_sent_ts_for_chain_synching;

                                        /*
                                         printf("CHAIN ACK %d->%d\n", (int)addr(),
                                         m_child_for_chain_synching);
                                         */
                                        memcpy((char*) data->data(), tpsn_data, sizeof(TpsnData));
                                        sendmsg(sizeof(TpsnData), data, m_child_for_chain_synching);

                                        m_child_for_chain_synching = -1;
                                        m_child_sent_ts_for_chain_synching = -1;

                                }

                                m_state = READY;
                        }
                        break;
                default:
                        break;
                }

                /*
                 hdr_cmn* packet = hdr_cmn::access(pckt);

                 printf("timestamp: %lf\n", packet->timestamp());
                 printf("tx_time: %lf\n", packet->txtime()); // send offset
                 printf("type: %d\n", packet->ptype());
                 */
        }

        Packet::free(pckt);
}

void TpsnUdpAgent::sendmsg(int nbytes, AppData* data, nsaddr_t daddr,
                const char *flags)
{
        if (m_alive)
        {
                TpsnData *tpsn_data = (TpsnData *) (((PacketData*) data)->data());
                ++m_sent_message_count[tpsn_data->type];

                if (m_max_node_in_cluster > 0)
                {
                        if (tpsn_data->type == SYNC_GROUP_ACK_PACKET || tpsn_data->type
                                        == SYNC_ACK_PACKET)
                        {
                                tpsn_data->group = m_group;
                                if ((int) addr() == 0)
                                {
                                        tpsn_data->level = m_max_depth - 1;
                                }
                                else
                                {
                                        tpsn_data->level = m_level;
                                }
                        }
                }
                //printf("Tpsn node sent message\n");
                Packet *p;
                int n;

                assert(size_> 0);

                n = nbytes / size_;

                if (nbytes == -1)
                {
                        printf("Error:  sendmsg() for UDP should not be -1\n");
                        return;
                }

                // If they are sending data, then it must fit within a single packet.
                if (data && nbytes > size_)
                {
                        printf("Error: data greater than maximum UDP packet size\n");
                        return;
                }

                double local_time = Scheduler::instance().clock();
                while (n-- > 0)
                {
                        p = allocpkt();
                        hdr_cmn::access(p)->size() = size_;
                        hdr_rtp* rh = hdr_rtp::access(p);
                        rh->flags() = 0;
                        rh->seqno() = ++seqno_;

                        hdr_ip* ih = hdr_ip::access(p);
                        ih->daddr() = daddr;

                        hdr_cmn::access(p)->timestamp() = (u_int32_t) (SAMPLERATE
                                        * local_time);
                        // add "beginning of talkspurt" labels (tcl/ex/test-rcvr.tcl)
                        if (flags && (0 == strcmp(flags, "NEW_BURST")))
                                rh->flags() |= RTP_M;
                        p->setdata(data);
                        target_->recv(p);
                }
                n = nbytes % size_;
                if (n > 0)
                {
                        p = allocpkt();
                        hdr_cmn::access(p)->size() = n;
                        hdr_rtp* rh = hdr_rtp::access(p);
                        rh->flags() = 0;
                        rh->seqno() = ++seqno_;

                        hdr_ip* ih = hdr_ip::access(p);
                        ih->daddr() = daddr;

                        hdr_cmn::access(p)->timestamp() = (u_int32_t) (SAMPLERATE
                                        * local_time);
                        // add "beginning of talkspurt" labels (tcl/ex/test-rcvr.tcl)
                        if (flags && (0 == strcmp(flags, "NEW_BURST")))
                                rh->flags() |= RTP_M;
                        p->setdata(data);
                        target_->recv(p);
                }
        }
        idle();
}

int TpsnUdpAgent::command(int argc, const char*const* argv)
{
        //printf("Tpsn node running command: %s\n", argv[1]);

        /*
         if (argc == 2 && (strcmp(argv[1], "sync") == 0))
         {
         TpsnData tpsn_data;
         m_state = SYNCING;
         tpsn_data.type = SYNC_PULSE_PACKET;
         tpsn_data.level = 0;
         tpsn_data.s_sent_ts = localClockCurrent();

         PacketData* data = new PacketData(sizeof(TpsnData));
         memcpy((char*)data->data(), &tpsn_data, sizeof(TpsnData));
         sendmsg(sizeof(TpsnData), data);
         return (TCL_OK);
         }
         */
        if (argc == 2 && (strcmp(argv[1], "setasroot") == 0))
        {
                TpsnData tpsn_data;
                m_state = READY;
                m_level = 0;
                m_group = 0;
                tpsn_data.type = LEVEL_DISCOVERY_PACKET;
                tpsn_data.level = m_max_depth - 1;
                tpsn_data.group = 0; // Root node will be a cluster alone
                tpsn_data.sender = (int) addr();
                tpsn_data.s_sent_ts = localClockCurrent();

                //printf("Using m_max_depth: %d\n", m_max_depth);
                //printf("%02d: Init: Setting level to: %d\n", (int)addr(), m_level);

                PacketData* data = new PacketData(sizeof(TpsnData));
                memcpy((char*) data->data(), &tpsn_data, sizeof(TpsnData));
                sendmsg(sizeof(TpsnData), data, IP_BROADCAST);
                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "initialise") == 0))
        {
                //printf("init\n");
                //printf("Interval before: %lf\n", m_interval);
                m_interval /= m_clock_drift_multiplier; // Because we use global scheduler
                //printf("Interval after: %lf\n", m_interval);
                m_tpsn_sync_timer.resched(m_interval);
                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "setasalive") == 0))
        {
                //printf("init\n");
                m_alive = true;
                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "setaszombie") == 0))
        {
                //printf("init\n");
                m_alive = false;
                return (TCL_OK);
        }
        if (argc == 2 && (strncmp(argv[1], "setmaxdepth", strlen("setmaxdepth"))
                        == 0))
        {
                const char *max_depth = &argv[1][strlen("setmaxdepth")];
                m_max_depth = atoi(max_depth);
                return (TCL_OK);
        }
        if (argc == 2 && (strncmp(argv[1], "setmaxchild", strlen("setmaxchild"))
                        == 0))
        {
                const char *max_child = &argv[1][strlen("setmaxchild")];
                m_max_child = atoi(max_child);

                if (m_max_child > 0)
                {
                        m_child_list = new ChildNode[m_max_child];
                }

                return (TCL_OK);
        }
        if (argc == 2 && (strncmp(argv[1], "setmaxnodeincluster", strlen("setmaxnodeincluster"))
                        == 0))
        {
                const char *max_node_in_cluster = &argv[1][strlen("setmaxnodeincluster")];
                m_max_node_in_cluster = atoi(max_node_in_cluster);

                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "enable_child_sync_timer") == 0))
        {
                //printf("init\n");
                m_child_sync_timer = true;
                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "local_clock") == 0))
        {
                int total_sent_message = 0;
                int total_received_message = 0;

                for (int i = 0; i < MESSAGE_TYPE_SIZE; ++i)
                {
                        total_received_message += m_received_message_count[i];
                        total_sent_message += m_sent_message_count[i];
                        //printf("%d:%d-%d\n", i, m_received_message_count[i], m_sent_message_count[i]);
                }

                //printf("Timeouts: %d\n", m_timeout_count);

                /*
                 printf("%02d: [L:%02d G:%02d P:%02d] Corrected time %lf (Ref: %lf)\n",
                 (int)addr(), m_level, m_group, (int)m_parent,
                 localClockCurrent(), (NOW * m_clock_drift_multiplier));
                 */
                printf("%02d 0 %02d %02d %02d %lf %lf %04d %04d\n",
                                (int) addr(), m_level, m_group, (int) m_parent,
                                localClockCurrent(), (NOW * m_clock_drift_multiplier),
                                total_sent_message, total_received_message);

                return (TCL_OK);
        }
        if (argc == 2 && (strcmp(argv[1], "time_complexity") == 0))
```

```
        {
                printf("%02d 0 %02d %02d %02d %lf %lf %d %lf\n",
                                (int) addr(), m_level, m_group, (int) m_parent,
                                m_init_time, m_inited_time, m_sync_fail_count, m_interval * m_clock_drift_multiplier);

                return (TCL_OK);
        }
        else
        {
                return (UdpAgent::command(argc, argv));
        }
}

double TpsnUdpAgent::estimatedSourceTimeOffset(double Ljt4, double Lit2,
                double Lit7, double Ljt6)
{
        return ((Ljt4 - Lit2) - (Lit7 - Ljt6)) / 2;
}

double TpsnUdpAgent::localClockCurrent()
{
        return (NOW * m_clock_drift_multiplier) + m_clock_offset;
}

double TpsnUdpAgent::localClock(double global_time)
{
        return (global_time * m_clock_drift_multiplier) + m_clock_offset;
}
bool TpsnUdpAgent::isMyChild(int child_node_id)
{
        for (int i = 0; i < m_child_count; ++i)
        {
                if (m_child_list[i].alive && m_child_list[i].node_id == child_node_id)
                {
                        return true;
                }
        }

        return false;
}

bool TpsnUdpAgent::setAsMyChild(int child_node_id, int descendant_count)
{
        if (m_child_count < m_max_child)
        {
                m_child_list[m_child_count].node_id = child_node_id;
                m_child_list[m_child_count].alive = true;
                m_child_list[m_child_count].descendant_count = descendant_count;
                ++m_child_count;

                return true;
        }

        return false;
}

bool TpsnUdpAgent::updateMyChild(int child_node_id, int descendant_count)
{
        for (int i = 0; i < m_child_count; ++i)
        {
                if (m_child_list[i].alive && m_child_list[i].node_id == child_node_id)
                {
                        m_child_list[i].descendant_count = descendant_count;
                        return true;
                }
        }

        return false;
}
bool TpsnUdpAgent::removeFromMyChildren(int child_node_id)
{
        for (int i = 0; i < m_child_count; ++i)
        {
                if (m_child_list[i].alive && m_child_list[i].node_id == child_node_id)
                {
                        --m_child_count;

                        if (m_child_count > 0)
                        {
                                m_child_list[i].alive = m_child_list[m_child_count].alive;
                                m_child_list[i].node_id = m_child_list[m_child_count].node_id;
                        }
                        else
                        {
                                m_child_list[i].alive = false;
                                m_child_list[i].node_id = 0;
                        }

                        return true;
                }
        }

        return false;
```

```
}

int TpsnUdpAgent::descendantCount()
{
        int result = 0;

        for (int i = 0; i < m_child_count; ++i)
        {
                result += m_child_list[i].descendant_count;
        }

        return result + m_child_count;
}

bool TpsnUdpAgent::disconnectNodeToFormNewCluster(int child_node_id)
{
        int other_nodes_count = 0;
        int child_node_index = -1;

        for (int i = 0; i < m_child_count; ++i)
        {
                if (m_child_list[i].node_id != child_node_id)
                {
                        other_nodes_count += m_child_list[i].descendant_count;
                        other_nodes_count += 1; // Add child
                }
                else
                {
                        child_node_index = i;
                }
        }
        other_nodes_count += 1; // Add cluster root

        if (child_node_index >= 0)
        {
                //printf("%d: other_nodes_count: %d\n", child_node_id, other_nodes_count);
                //printf("%d: descendant_count: %d\n", child_node_id, m_child_list[child_node_index].descendant_count);
                return (other_nodes_count >= m_max_node_in_cluster
                                && m_child_list[child_node_index].descendant_count
                                                >= m_max_node_in_cluster);
        }
        else
        {
                return false;
        }
}

bool TpsnUdpAgent::connectNodeToCluster(int child_node_id)
{
        int other_nodes_count = 0;
        int child_node_index = -1;

        for (int i = 0; i < m_child_count; ++i)
        {
                if (m_child_list[i].node_id != child_node_id)
                {
                        other_nodes_count += m_child_list[i].descendant_count;
                        other_nodes_count += 1; // Add child
                }
                else
                {
                        child_node_index = i;
                }
        }
        other_nodes_count += 1; // Add cluster root

        if (child_node_index >= 0)
        {
                //printf("%d: other_nodes_count: %d\n", child_node_id, other_nodes_count);
                //printf("%d: descendant_count: %d\n", child_node_id, m_child_list[child_node_index].descendant_count);
                return (m_child_list[child_node_index].descendant_count == 0
                                || (m_child_list[child_node_index].descendant_count
                                                > m_max_node_in_cluster && other_nodes_count
                                                < m_max_node_in_cluster));
        }
        else
        {
                return false;
        }
}
```

Table B.1. Reference TPSN Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	307	308
1	0	6	0	48	9636.31	11266.43	220	223
2	0	12	0	5	9638.92	9341.3	94	97
3	0	15	0	49	9644.21	10845.34	202	205
4	0	10	0	22	9624.43	8644.21	87	88
5	0	11	0	18	9641.66	10557.07	378	380
6	0	8	0	9	9628.49	9404.6	189	194
7	0	5	0	36	9640.33	10772.7	108	115
8	0	5	0	36	9638.57	10199.24	102	106
9	0	7	0	29	9626.21	9074.93	384	390
10	0	13	0	21	9646.42	10973.58	496	503
11	0	5	0	41	9631.05	9672.16	193	196
12	0	7	0	1	9654.28	10993.48	109	111
13	0	16	0	3	9643.49	9361.29	94	97
14	0	10	0	34	9616.6	8395.73	84	87
15	0	6	0	42	9638.94	9332.19	93	95
16	0	14	0	10	9648.54	11427.32	114	119
17	0	14	0	10	9628.14	7124.93	71	77
18	0	10	0	34	9632.01	10313.83	395	397
19	0	14	0	10	9646.27	10679.13	107	111
20	0	7	0	38	9652.93	10433.73	297	302
21	0	12	0	5	9639.19	9455.77	305	309
22	0	9	0	45	9628.13	9465.85	277	279
23	0	7	0	29	9634.05	10489.06	105	111
24	0	4	0	0	9636.2	10649.9	106	107
25	0	6	0	42	9654.61	11008.82	110	112
26	0	7	0	29	9632.03	10727.9	108	114
27	0	7	0	29	9631.68	9706.84	97	105
28	0	9	0	46	9639.07	9003.69	90	94
29	0	6	0	11	9631.51	9680.3	762	768
30	0	10	0	22	9630.55	9818.35	98	99
31	0	7	0	29	9632.12	10001.19	100	107
32	0	8	0	9	9632.14	9787.78	98	106
33	0	11	0	18	9636.5	9921.8	99	100
34	0	9	0	6	9629.18	9549.99	281	284
35	0	11	0	18	9630.09	9097.28	91	91
36	0	4	0	0	9640.02	10717.88	315	316
37	0	12	0	5	9628.84	8721.34	87	94
38	0	6	0	42	9642.57	11730.43	221	224
39	0	13	0	21	9638.39	10312.96	103	111
40	0	8	0	20	9653.07	10467.73	105	110
41	0	4	0	0	9631.2	9541	394	398
42	0	5	0	41	9641.98	11427.92	432	437
43	0	7	0	29	9631.19	7829.53	79	87
44	0	7	0	29	9628.96	9304.36	196	204
45	0	8	0	44	9636.2	10440.97	197	200
46	0	8	0	20	9650.01	8923.68	179	183
47	0	8	0	9	9633.83	10548.97	105	110
48	0	5	0	41	9627.37	9011.25	201	203
49	0	14	0	10	9643.19	9882	207	212

Table B.2. Clustered Synchronization Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	153	154
1	0	2	1	48	9635.37	11266.43	220	223
2	0	0	3	5	9630	9341.3	47	50
3	0	3	3	49	9642.19	10845.34	185	188
4	0	2	2	22	9633.75	8644.21	87	88
5	0	3	2	18	9628.85	10557.07	284	286
6	0	0	2	9	9625.58	9404.6	152	157
7	0	1	1	36	9640.33	10772.7	108	115
8	0	1	1	36	9638.57	10199.24	102	106
9	0	3	1	29	9626.38	9074.93	271	277
10	0	1	3	21	9629.67	10973.58	504	511
11	0	1	1	41	9631.4	9672.16	236	239
12	0	3	1	1	9653.33	10993.48	109	111
13	0	0	4	3	9625.92	9361.29	47	50
14	0	2	2	34	9616.02	8395.73	84	87
15	0	2	1	42	9629.19	9332.19	93	95
16	0	2	3	10	9641.53	11427.32	114	119
17	0	2	3	10	9610.38	7124.93	71	77
18	0	2	2	34	9628.31	10313.83	467	469
19	0	2	3	10	9629.51	10679.13	107	111
20	0	3	1	38	9644.43	10433.73	229	234
21	0	0	3	5	9626.38	9455.77	260	264
22	0	1	2	45	9637.44	9465.85	277	279
23	0	3	1	29	9633.47	10489.06	105	111
24	0	0	1	0	9645.76	10649.9	53	54
25	0	2	1	42	9644.86	11008.82	110	112
26	0	3	1	29	9631.44	10727.9	108	114
27	0	3	1	29	9631.1	9706.84	97	105
28	0	1	2	46	9629.71	9003.69	90	94
29	0	2	1	11	9630.93	9680.3	841	847
30	0	2	2	22	9639.86	9818.35	98	99
31	0	3	1	29	9631.54	10001.19	100	107
32	0	0	2	9	9633.01	9787.78	49	57
33	0	3	2	18	9632.05	9921.8	99	100
34	0	1	2	6	9626.59	9549.99	320	323
35	0	3	2	18	9626.39	9097.28	91	91
36	0	0	1	0	9640.02	10717.88	262	263
37	0	0	3	5	9620.81	8721.34	44	51
38	0	2	1	42	9649.64	11730.43	268	271
39	0	1	3	21	9624.57	10312.96	103	111
40	0	0	2	20	9644.57	10467.73	53	58
41	0	0	1	0	9630.25	9541	371	375
42	0	1	1	41	9631.29	11427.92	465	470
43	0	3	1	29	9630.61	7829.53	79	87
44	0	3	1	29	9627.8	9304.36	155	163
45	0	0	2	44	9645.52	10440.97	145	148
46	0	0	2	20	9629.44	8923.68	134	138
47	0	0	2	9	9634	10548.97	53	58
48	0	1	1	41	9627.37	9011.25	201	203
49	0	2	3	10	9631.96	9882	243	248

Table B.3. Chain Synchronization Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	153	154
1	0	2	1	48	9627.01	11266.43	227	230
2	0	0	3	5	9626.53	9341.3	47	50
3	0	3	3	49	9633.73	10845.34	185	188
4	0	2	2	22	9634.18	8644.21	87	88
5	0	3	2	18	9629.58	10557.07	284	286
6	0	0	2	9	9626.24	9404.6	167	172
7	0	1	1	36	9640.33	10772.7	108	115
8	0	1	1	36	9638.57	10199.24	102	106
9	0	3	1	29	9626.75	9074.93	271	277
10	0	1	3	21	9627.93	10973.58	537	544
11	0	1	1	41	9631.24	9672.16	305	308
12	0	3	1	1	9644.98	10993.48	109	111
13	0	0	4	3	9621.23	9361.29	47	50
14	0	2	2	34	9616.83	8395.73	84	87
15	0	2	1	42	9628.4	9332.19	93	95
16	0	2	3	10	9636.84	11427.32	114	119
17	0	2	3	10	9605.7	7124.93	71	77
18	0	2	2	34	9629.03	10313.83	495	497
19	0	2	3	10	9627.77	10679.13	107	111
20	0	3	1	38	9643.64	10433.73	229	234
21	0	0	3	5	9627.11	9455.77	289	293
22	0	1	2	45	9637.88	9465.85	287	289
23	0	3	1	29	9633.8	10489.06	105	111
24	0	0	1	0	9645.76	10649.9	53	54
25	0	2	1	42	9644.07	11008.82	110	112
26	0	3	1	29	9631.78	10727.9	108	114
27	0	3	1	29	9631.46	9706.84	97	105
28	0	1	2	46	9626.04	9003.69	90	94
29	0	2	1	11	9631.26	9680.3	900	906
30	0	2	2	22	9640.3	9818.35	98	99
31	0	3	1	29	9631.87	10001.19	100	107
32	0	0	2	9	9633.67	9787.78	49	57
33	0	3	2	18	9630.09	9921.8	99	100
34	0	1	2	6	9627.32	9549.99	363	366
35	0	3	2	18	9627.12	9097.28	91	91
36	0	0	1	0	9640.02	10717.88	262	263
37	0	0	3	5	9621.87	8721.34	44	51
38	0	2	1	42	9648.85	11730.43	274	277
39	0	1	3	21	9625.11	10312.96	103	111
40	0	0	2	20	9643.77	10467.73	53	58
41	0	0	1	0	9630.25	9541	416	420
42	0	1	1	41	9630.5	11427.92	487	492
43	0	3	1	29	9630.94	7829.53	79	87
44	0	3	1	29	9627.14	9304.36	155	163
45	0	0	2	44	9646.18	10440.97	155	158
46	0	0	2	20	9625.77	8923.68	134	138
47	0	0	2	9	9634.36	10548.97	53	58
48	0	1	1	41	9625.1	9011.25	227	229
49	0	2	3	10	9626.29	9882	247	252

Table B.4. Adaptive Synchronization Interval Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	188	189
1	0	2	1	48	9634.92	11266.43	261	264
2	0	0	3	5	9623.08	9341.3	41	44
3	0	3	3	49	9632.64	10845.34	162	165
4	0	2	2	22	9632.47	8644.21	85	86
5	0	3	2	18	9634.15	10557.07	252	254
6	0	0	2	9	9623.13	9404.6	119	124
7	0	1	1	36	9635.17	10772.7	119	126
8	0	1	1	36	9639.1	10199.24	99	103
9	0	3	1	29	9626.49	9074.93	247	253
10	0	1	3	21	9633.35	10973.58	593	600
11	0	1	1	41	9629.57	9672.16	202	205
12	0	3	1	1	9634.08	10993.48	125	127
13	0	0	4	3	9620.85	9361.29	41	44
14	0	2	2	34	9620.14	8395.73	106	109
15	0	2	1	42	9628.97	9332.19	74	76
16	0	2	3	10	9633.37	11427.32	153	158
17	0	2	3	10	9625.71	7124.93	115	121
18	0	2	2	34	9632.77	10313.83	377	379
19	0	2	3	10	9632.33	10679.13	105	109
20	0	3	1	38	9636.83	10433.73	230	235
21	0	0	3	5	9628.02	9455.77	264	268
22	0	1	2	45	9634.66	9465.85	236	238
23	0	3	1	29	9630.58	10489.06	90	96
24	0	0	1	0	9638.77	10649.9	74	75
25	0	2	1	42	9639.84	11008.82	128	130
26	0	3	1	29	9635.68	10727.9	107	113
27	0	3	1	29	9629.91	9706.84	69	77
28	0	1	2	46	9627.13	9003.69	86	90
29	0	2	1	11	9629.58	9680.3	763	769
30	0	2	2	22	9639.29	9818.35	70	71
31	0	3	1	29	9636.16	10001.19	70	77
32	0	0	2	9	9630.25	9787.78	41	49
33	0	3	2	18	9636.75	9921.8	74	75
34	0	1	2	6	9626	9549.99	297	300
35	0	3	2	18	9622	9097.28	75	75
36	0	0	1	0	9635.1	10717.88	292	293
37	0	0	3	5	9625.92	8721.34	55	62
38	0	2	1	42	9642.51	11730.43	295	298
39	0	1	3	21	9630.38	10312.96	87	95
40	0	0	2	20	9639.7	10467.73	62	67
41	0	0	1	0	9630.6	9541	362	366
42	0	1	1	41	9641.18	11427.92	544	549
43	0	3	1	29	9625.71	7829.53	132	140
44	0	3	1	29	9625.94	9304.36	150	158
45	0	0	2	44	9637.25	10440.97	150	153
46	0	0	2	20	9630.22	8923.68	138	142
47	0	0	2	9	9632.49	10548.97	71	76
48	0	1	1	41	9626.54	9011.25	212	214
49	0	2	3	10	9629.08	9882	206	211

Table B.5. Balanced Children Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	188	189
1	0	2	1	48	9634.92	11266.43	261	264
2	0	0	3	5	9624.81	9341.3	42	45
3	0	3	3	49	9634.34	10845.34	168	171
4	0	2	2	22	9626.51	8644.21	86	87
5	0	3	2	18	9624.93	10557.07	262	264
6	0	0	2	9	9624.82	9404.6	128	133
7	0	1	1	36	9635.17	10772.7	122	129
8	0	1	1	36	9639.1	10199.24	100	104
9	0	0	2	26	9628.77	9074.93	201	218
10	0	1	3	21	9632.3	10973.58	601	608
11	0	1	1	41	9630.68	9672.16	190	193
12	0	3	1	1	9634.08	10993.48	125	127
13	0	0	4	3	9626.71	9361.29	43	46
14	0	2	2	34	9621.67	8395.73	107	110
15	0	2	1	42	9628.59	9332.19	74	76
16	0	2	3	10	9633.22	11427.32	155	160
17	0	2	3	10	9628.73	7124.93	115	121
18	0	2	2	34	9624.9	10313.83	386	388
19	0	2	3	10	9628.8	10679.13	105	109
20	0	3	1	38	9636.6	10433.73	231	236
21	0	0	3	5	9630.89	9455.77	266	270
22	0	1	2	45	9633.78	9465.85	238	240
23	0	3	1	29	9631.77	10489.06	141	147
24	0	0	1	0	9638.77	10649.9	74	75
25	0	2	1	42	9639	11008.82	128	130
26	0	3	1	29	9634.24	10727.9	226	232
27	0	3	1	29	9631	9706.84	72	80
28	0	1	2	46	9631.56	9003.69	87	91
29	0	2	1	11	9630.77	9680.3	482	491
30	0	2	2	22	9634.71	9818.35	71	72
31	0	0	2	23	9631.17	10001.19	47	64
32	0	0	2	9	9626.32	9787.78	44	52
33	0	3	2	18	9632.42	9921.8	73	74
34	0	1	2	6	9624.83	9549.99	303	306
35	0	3	2	18	9621.45	9097.28	75	75
36	0	0	1	0	9635.1	10717.88	292	293
37	0	0	3	5	9621.02	8721.34	54	61
38	0	2	1	42	9640.97	11730.43	296	299
39	0	1	3	21	9632.3	10312.96	88	96
40	0	0	2	20	9639.23	10467.73	62	67
41	0	0	1	0	9630.6	9541	360	364
42	0	1	1	41	9639.95	11427.92	545	550
43	0	0	2	26	9610.34	7829.53	53	71
44	0	3	1	29	9628.01	9304.36	153	161
45	0	0	2	44	9636.35	10440.97	152	155
46	0	0	2	20	9631.51	8923.68	139	143
47	0	0	2	9	9631.43	10548.97	70	75
48	0	1	1	41	9626.54	9011.25	212	214
49	0	2	3	10	9629.39	9882	216	221

Table B.6. Balanced Cluster Results

Node	StartClk	Lvl	Grp	Prnt	SyncClk	Clk	MsgSent	MsgRecv
0	0	0	0	0	9631.3	9631.3	188	189
1	0	2	1	48	9634.92	11266.43	261	264
2	0	1	4	5	9631.47	9341.3	82	85
3	0	3	5	49	9634.32	10845.34	163	166
4	0	2	3	22	9625.57	8644.21	86	87
5	0	0	4	18	9635.03	10557.07	273	275
6	0	0	3	9	9630.38	9404.6	128	133
7	0	1	1	36	9635.17	10772.7	122	129
8	0	1	1	36	9639.1	10199.24	100	104
9	0	3	2	26	9627.93	9074.93	256	273
10	0	1	5	21	9628.6	10973.58	581	588
11	0	0	2	41	9631.16	9672.16	135	138
12	0	3	1	1	9634.08	10993.48	125	127
13	0	0	6	3	9628.52	9361.29	45	48
14	0	2	3	34	9626.32	8395.73	107	110
15	0	2	1	42	9628.59	9332.19	74	76
16	0	2	5	10	9630.48	11427.32	147	152
17	0	2	5	10	9618	7124.93	108	114
18	0	2	3	34	9634.35	10313.83	315	317
19	0	2	5	10	9628.24	10679.13	102	106
20	0	3	1	38	9636.6	10433.73	231	236
21	0	0	5	5	9633.34	9455.77	266	270
22	0	1	3	45	9632.67	9465.85	238	240
23	0	2	2	29	9635.77	10489.06	167	173
24	0	0	1	0	9638.77	10649.9	74	75
25	0	2	1	42	9639	11008.82	128	130
26	0	2	2	29	9640.01	10727.9	320	326
27	0	2	2	29	9631.53	9706.84	72	80
28	0	1	2	46	9631.56	9003.69	87	91
29	0	1	2	11	9631.31	9680.3	465	474
30	0	2	3	22	9638.53	9818.35	70	71
31	0	3	2	23	9635.11	10001.19	73	90
32	0	0	3	9	9635.96	9787.78	47	55
33	0	3	3	18	9632.64	9921.8	70	71
34	0	1	3	6	9631.02	9549.99	293	296
35	0	3	3	18	9631.33	9097.28	75	75
36	0	0	1	0	9635.1	10717.88	292	293
37	0	1	4	5	9634.45	8721.34	83	90
38	0	2	1	42	9640.97	11730.43	296	299
39	0	1	5	21	9631.97	10312.96	87	95
40	0	0	2	20	9639.23	10467.73	62	67
41	0	0	1	0	9630.6	9541	322	326
42	0	1	1	41	9639.95	11427.92	545	550
43	0	3	2	26	9636.56	7829.53	100	118
44	0	2	2	29	9628.83	9304.36	152	160
45	0	0	3	44	9636.32	10440.97	153	156
46	0	0	2	20	9631.51	8923.68	139	143
47	0	0	3	9	9645.28	10548.97	73	78
48	0	1	1	41	9626.54	9011.25	212	214
49	0	2	5	10	9632.3	9882	209	214