A SIMULATION FRAMEWORK TO ENSURE DATA CONSISTENCY IN SENSOR NETWORKS

by

NIKHIL JEEVANLAL SHAH

B.E., University of Pune, 2005

A REPORT

submitted in partial fulfillment of the requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2008

Approved by:

Major Professor
Dr. Gurdip Singh

Abstract

The objective of this project is to address the problem of data consistency in sensor network applications. An application may involve data being gathered from several sources to be delivered to multiple sinks, resulting in multiple data streams with several sources and sinks for each stream. There may be several inter-stream constraints to be satisfied in order to ensure data consistency. In this report, we model this problem as that of variable sharing between the components in an application, and propose a framework for implementing variable sharing in a distributed sensor network. In this framework, we define the notion of variable sharing in component based systems. We allow the application designer to specify data consistency constraints in an application. Given an application, we implement a tool to identify various types of shared variables in an application. Given the shared variables and the data consistency constraints, we provide an infrastructure to implement the shared variables. This infrastructure has tools to synthesize the code to be deployed on each of the nodes in the physical topology. The infrastructure has been built for the TinyOS platform. We have evaluated the framework using several examples using the TOSSIM simulator.

Table of Contents

List of Figures iv
Acknowledgements v
CHAPTER 1 - Introduction 1
CHAPTER 2 - Project Description 5
2.1 Specification of Components in NesC 6
2.2 Specification of a Configuration in NesC 7
2.3 What is Variable Sharing? 8
2.4 *AppGraph.xml* - The Input Application Graph File 9
2.2.1 Application Graph Parameters 9
2.3 Network Topology 11
2.3.1 Topology Connectivity 11
2.5 Mapping of application components to network nodes 12
CHAPTER 3 - Labeling Algorithm 14
3.1 Identifying Shared Variables 15
3.2 Building an adjacency list for the network topology 17
3.3 Identifying the Accumulator and Distributor Nodes 18
3.4 Analyzing Various Consistency Constraints 19
CHAPTER 4 - Consistency Framework 21
4.1 *specs.txt* File Format 21
4.2 Consistency Framework Implementation 22
CHAPTER 5 - Performance Results 25
References 39

List of Figures

Figure 1.1 Architecture of the Components of the Framework 3
Figure 2.1 Examples of various types of Components 6
Figure 2.2 Sample component code from Blink.nc file 6
Figure 2.3 Example of a simple Blink Application 7
Figure 2.4 Example of a simple Blink.nc configuration files 8
Figure 2.6 *AppGraph.xml* file with input wiring information 9
Figure 2.7 *topo.txt* file with Input Topology graph information 11
Figure 2.8 *mapping.txt* file with Input Mapping information 12
Figure 2.9 *causal.txt* file with Causal Consistency Constraints 13
Figure 3.1 Examples of variable sharing 14
Figure 3.2 Examples of variable sharing 16
Figure 3.3 Application Graph Example 1 16
Figure 3.4 Network Graph Example 1 18
Figure 4.1 Specification File structure Example 1 21
Figure 5.1 Application Graph Scenario 1 25
Figure 5.2 Network Graph Scenario 1 26
Figure 5.3 Generated *specs.txt* file for Scenario 1 26
Figure 5.4 Application Graph Scenario 2 28
Figure 5.5 Network Graph Scenario 2 29
Figure 5.6 Generated *specs.txt* file for Scenario 2 29
Figure 5.7 output for Scenario 2 31
Figure 5.8 Tabular representation of Performance Results 1 33
Figure 5.9 *specs.txt* file generated for Scenario 3 35
Figure 5.10 Generated *specs.txt* file for Scenario 4 36
Figure 5.11 Tabular representation of Performance Results 37

Acknowledgements

I would like to take this opportunity to express my gratitude to some important people who have inspired and guided me to complete this project.

Firstly, I thank my Major Professor and Advisor, Prof. Gurdip Singh for his continued support and guidance throughout this project. I thank you for your patience, for always being willing to point me in the right direction.

I thank my committee members, Prof. Daniel Andersen and Prof. John Hatcliff for their support.

Finally, I would like to thank my family and friends for their unrelenting support and encouragement, and for raising my spirits whenever I needed it.

CHAPTER 1 - Introduction

Sensor network applications are often formed by putting together different components to form high level configurations. A configuration is used to assemble components together by connecting the interfaces used by components to interfaces provided by others. The structure of most of the sensor network applications involves, various sensor nodes sensing attributes such as temperature and humidity which are then propagated to different nodes where these values are processed and finally collected at sink node for further analysis and possible actions. We can see most of these applications involve data flowing between different components.

The TinyOS system, libraries, and applications are written in NesC, a new language for programming structured component-based applications. The NesC language is primarily intended for embedded systems such as sensor networks. It has a C-like syntax, but supports the TinyOS concurrency model, as well as mechanisms for structuring, naming, and linking together software components into robust network embedded systems. The principal goal is to allow application designers to build components that can be easily composed into complete, concurrent systems. NesC applications are built out of **components** with well-defined, bidirectional **interfaces**

In NesC, a sensor network application is designed by composing one or more components whose ports are linked together. A component is defined by a set of ports, where each port may offer a set of interfaces. Each interface has an operation and an event associated with it. A sensor application may involve data being gathered from several sources to be delivered to multiple sinks, resulting in multiple data streams with several sources and sinks for each stream. There may be several inter-stream constraints to be satisfied in order to ensure data consistency.

This paper addresses the problem of data consistency in sensor networks. We consider networks in which there are three types of nodes: those that are producing data (writers), those that are consuming data (readers) and intermediate nodes which consume data and then produce data (performing intermediate computations). A significant amount of research has been done in developing algorithms/protocols for data collection and dissemination in sensor

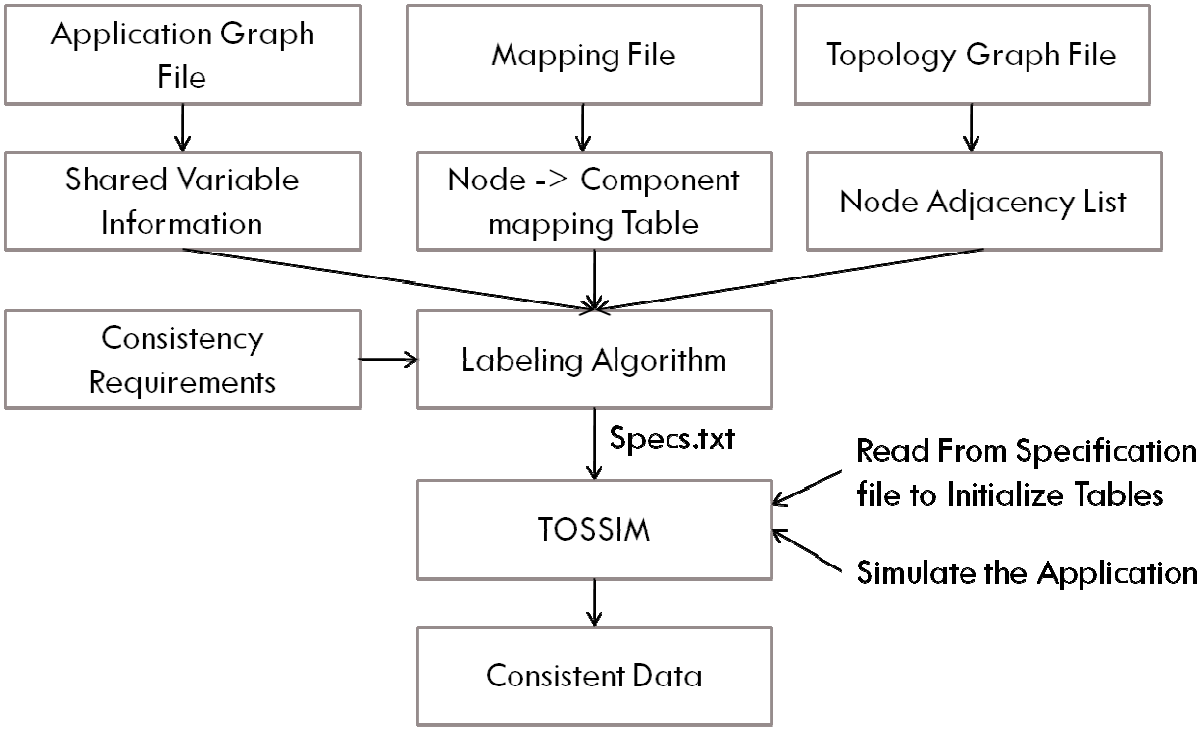
networks. A large body of this work has focused on mechanisms to ensure least amount of energy consumed (such as setting up shortest paths from producers to consumers, and to share/aggregate data in-transit to reduce the amount of data propagated). There has also been work done to address data consistency issues. However, this work has mainly been in the context of a single stream (or same data). For example, there has been work on addressing QoS issues such as latency, freshness of data, error probability, and approximation. These issues are desirable and relevant in cases where independent users are accessing the sensor networks for information, and each user may have different constraints.

Query mechanisms have been proposed which allow one to specify queries which get data from a particular sensor or sensors in a specific area. Researchers have looked at how overlap between different queries can be exploited to reduce the amount of information propagation.

The problem of data consistency across data streams due to constraints within the same application has not been addressed. This problem, similar to distributed shared memory, has been studied extensively in distributed computing. However, problems specific to sensor networks have not been looked at. In particular, in the standard DSM, all data is treated in the same way (that is, all data is required to be consistent). In the case of sensor networks, only some data may have to be delivered in a consistent manner. .

In this report, we model this problem as that of variable sharing between the components in an application, and propose a framework for implementing variable sharing in a distributed sensor network. Fig 1.1 shows the architecture diagram for the proposed framework.

Figure 1.1 Architecture of the Components of the Framework



The first stage processes the various input files and extracts the necessary information. The Application Graph represented in the form of AppGraph.xml file is analyzed to build the shared variable information table. The Network/Topology graph file, topo.txt, is analyzed to build the node adjacency list. The component to node mapping information given in the file, mapping.txt, is used to build a table containing the component- to node mapping. Various consistency requirements such as causal consistency, atomic consistency are specified in the files, causal.txt and atomic.txt, respectively which are used to build the consistency mechanisms.

All of the information obtained from the various files is given as input to the Labeling Algorithm which then produces a file, *specs.txt*, which contains information on how messages for each identified shared variable are to be processed. This specs.txt file is then used by the Consistency Framework to satisfy various consistency requirements.

This report is organized as follows -

1. Chapter 2 explains the following files: *AppGraph.xml* file which is the file users use to specify application graph information, *topo.txt* file users use to specify network topology

information, *mapping.txt* file users use to specify mapping information between application graph and topology graph, *causal.txt* and *atomic.txt* specifying causal and atomic consistency constraints.

2. Chapter 3 explains the Labeling Framework which is the most important component of this framework.

3. Chapter 4 explains *specs.txt* file generated by Labeling framework that NesC consistency simulation framework uses to initialize share variable information tables associated with each node. And explains how it can be used as a part of execution of applications using this framework.

CHAPTER 2 - Project Description

This chapter aims to give a detailed description of how the different features of the Labeling framework have been implemented. The description will be divided into the following categories:

- Specification of components and configurations and how we define variable sharing in a given application configuration.

- Description of the "Application Graph File" that contains all information regarding a configuration including the wiring information describing the interconnection of the ports of different components.

- Description of the "Topology Graph File" containing information about how the different nodes in the physical topology are connected with each other.

- Description of the "Mapping File" containing mapping information of various components onto various nodes.

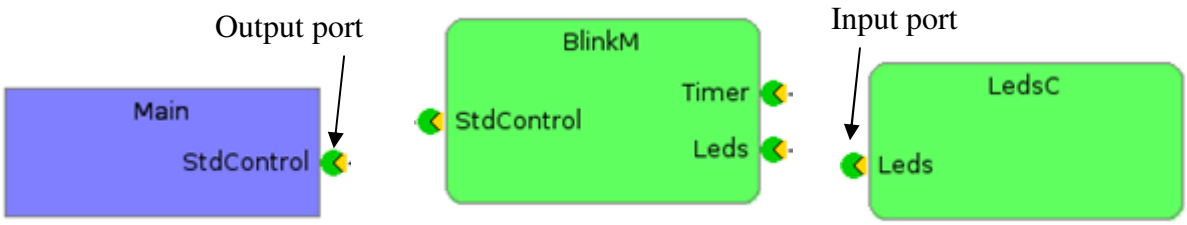
- Description of the "Causal Consistency constraints File" containing information of the causal consistency constraints to be imposed between different ports of a component.

- Description of the "Atomic Consistency constraints File" containing names of all the composite components requiring atomic execution.

2.1 Specification of Components in NesC

An application consists of a set of components. Each component has a set of input and output ports. We now give some examples of different types of components

Figure 2.1 Examples of various types of Components



(I) (II) (III)

Component Main shown in Fig 2.1(I) has only one provides (output) port *stdcontrol*. Such component acts as a writer component. Component 'BlinkM' shown in Fig 2.1(II) has one uses (input) port *stdcontrol* and two provides (output) ports *Timer and Leds* respectively. Such a component serves as an intermediate component which reads values written on its input port, processes them and makes it available on an output port. 'LedsC' component shown in Fig 2.1(III) has one uses (input) port *Leds*. Such a component acts as a reader component. Fig 2.2 shows a sample NesC code for component BlinkM.

Figure 2.2 Sample component code from Blink.nc file

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
```

2.2 Specification of a Configuration in NesC

A configuration is a collection of components. Configurations are used to build applications by assemblying a set of components together and connecting interfaces used by components to interfaces provided by other components. This interconnection of prts is called **wiring.** A configuration can have nested sub-configurations. A port of a configuration can be equated to port of a component inside it.

Now we will see an example of how does a configuration looks like and how various components are wired together to form an application using a sample NesC configuration file.

Figure 2.3 Example of a simple Blink Application

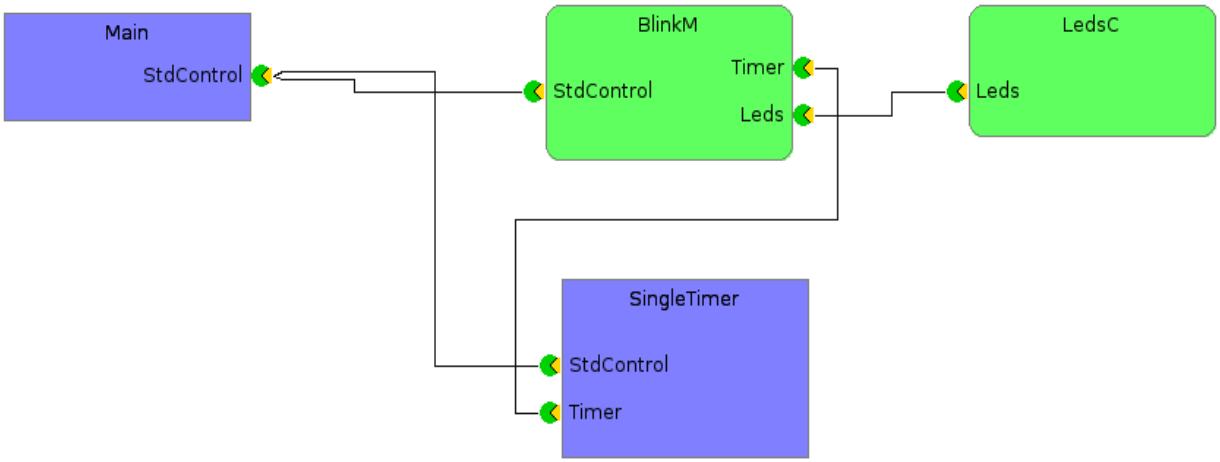


Fig. 2.3 shows us an example of a simple Blink application where different components are wired together to form a configuration as shown. The purpose of this application is to blink the led whenever a timer gets fired. Fig.2.4 shows a sample Blink.nc configuration file.

Figure 2.4 Example of a simple Blink.nc configuration files

```
Blink.nc configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

2.3 What is Variable Sharing?

Now that we have looked at components and configurations, we will see how we define variable sharing between them. Each component C has a set, C.in, of input ports and a set , C.out, of output ports. Each input port p in C.in is associated with a variable, p.var, which is read by C. Each output port p is also associated with a variable p.var that is written by C.

As we saw earlier, a configuration is defined by a set of components and wiring between the ports of the components. Data sharing is specified by the wiring between the components. When an output port, p1, of a component C1 is connected to input port p2 of component C2, then C1.p1.var and C2.p2.var are mapped as a single shared variable. A configuration may also have input ports and output ports. An input port of a configuration Conf can be bound to an input port of one or more components. In this case again, all of these input ports are mapped to the same variable that is associated with the input port of the configuration. Similarly, the output port of a configuration can be bound to the output ports of more than one component, and in this case as well, all of these variables are mapped to the same shared variable written by the components.

Fig 2.5 shows examples of variable sharing. Fig 2.5(1) shows wiring between a writer and a reader component. As these components share a variable between them we say that C2.P2 reads from C1.P1. In Fig 2.5(2) as we can see that we have configuration Conf with nested sub-configuration C3 where port P1 of Conf can be equated to port P3 of component C3. So we can say that C3.P3 reads from C2.P2. Similarly, in Fig 2.5(3) port P3 of C3 is equated with P1 of Conf.

2.4 *AppGraph.xml* - The Input Application Graph File

We represent sensor network application information in a file, AppGraph.xml, in the XML format. This contains information about input/output ports available for each component and its connection with other ports.

2.2.1 Application Graph Parameters

Fig. 2.4 illustrates the format of the application graph parameters which can be specified in *AppGraph.xml*. Following is an example of AppGraph.xml file of the configuration shown in Fig. 2.3

Figure 2.5 ***AppGraph.xml*** **file with input wiring information**

```
<Info>
  <components>
      <HOME>
      <name>D</name>
      <input_ports>
        <wire>
         <port>P2</port>
          <connected_to>
            <wire_to>P1</wire_to>
            <node>C</node>
          </connected_to>
          <equated_to>
            <wire_to>P4</wire_to>
            <node>E</node>
          </equated_to>
         <equated_to>
            <wire_to>P5</wire_to>
            <node>F</node>
         </equated_to>
        </wire>
      </input_ports>
```

```
    <output_ports>
        <wire>
        <port>P1</port>
         <connected_to>
            <wire_to>P2</wire_to>
            <node>D</node>
          </connected_to>
        </wire>
      </output_ports>
     <nested_comp>
      <name>G</name>
      <name>H</name>
     </nested_comp>
    </HOME>
  </components>
  <commands>
  </commands>
</Info>
```

The following points explain the different XML tags in the AppGraph.xml file from Fig. 2.6 –

1. The tag Name specifies the name of the component
2. The tag output_ports specifies all of the ports which are *provided by* the component
3. The tag input_ports specifies all of the ports *used by* the component
4. The tag Port specifies the port name of the current component defined by name tag, the tag connected_to specifies the component name and the tag wire specifies the port name it is connected to .For example, the connection A.P1 -> C.P4 is expressed as follows (multiple wiring's are similarly added by adding new wire tags)

```
<name>A</name>
   <output_ports>
     <wire>
     <port>P1</port>
      <connected_to>
         <wire_to>P2</wire_to>
         <node>D</node>
       </connected_to>
     </wire>
   </output_ports>
```

5. The tag equated_to specifies wiring information of composite component which can be equated to nested components within it

6. The tag nested_comp specifies list of nested sub-components present within a component

2.3 Network Topology

The *file topo.txt* is used to describe the topology of the network on which the application is to be deployed. The network topology is specified using the following parameters:

- Connectivity between the nodes,
- Coordinates of each of the nodes.

Both of these pieces of information are related. Given the coordinates of the nodes and the communication model used, one can determine the pair of nodes which can communicate with each other. At present, we require this information to be specified separately.

2.3.1 Topology Connectivity

Fig. 2.5 illustrates the format of the topology file, *topo.txt,* which specifies the connectivity between the nodes. .

Figure 2.6 *topo.txt* file with Input Topology graph information

numnodes 8
1 3 5 7
2 3
3 5 6
4 6
6 7

The following points explain the format of the file:

1. The number of nodes in the actual network is specified by the keyword *numnodes*.
2. The node connectivity of the network is specified as follows –
   i. The first parameter is id of the node whose connectivity information is being provided (e.g. the first line is specifying information about node 1).
   ii. The subsequent numbers on the same line indicate the ids of the nodes which are connected with the current node e.g. the first line of topo.txt file specifies that

node 3, node 5 and node 7 are connected with node 1. Similarly, each new line contains connectivity information of that node.

2.5 Mapping of application components to network nodes

To deploy an application on a network topology, we need to know the nodes where each of the application components is to be deployed. We assume that this information is provided by the application designer. This is specified in the file, *mapping.txt.* When developing an application in Cadena, it is possible to associate an attribute, location, with each component, and the designer can specify the locations when designing the application. This information is stored in an internal Cadena data structure along with the application scenario information. It is possible to derive this information (and hence, construct the file mapping.txt) from the application information. At present, we assume that this file is available to our tool. We require that at most one component is mapped to each node. Fig. 2.6 illustrates the format of the mapping file parameters that can be specified in *mapping.txt.*

Figure 2.7 *mapping.txt* file with Input Mapping information

numcomponents 4
a 1
b 2
c 4
d 6

The following points explain the format of the file *mapping.txt* from Fig. 2.8 –

1. The number of components in the application graph are specified by the keyword *numcomponents*.

2. The component - node mapping can be specified as follows –
   i. First parameter is the component name which is mapped onto node Number specified in second parameter e.g. component ‘a’ is mapped onto node number 1

2.6 *Causal.txt* - Causal Consistency Constraints

Each component can specify the consistency requirements for its incoming data. Causal consistency constraint is defined as follows. Let p1 and p2 be uses ports of a component C.

Causal delivery (p1, p2): Let e1 and e2 be two events (containing values v1 and v2 respectively) delivered on p1 and p2 respectively. Then, there must exists a value v1' such that v1' is more recent than v1 and v2 depends on v1'.

This information can be specified in *causal.txt* file. The format of causal.txt file is as shown in Fig 2.9.

Figure 2.8 *causal.txt* file with Causal Consistency Constraints

C P4 P5

The causal consistency constraints between different input ports of a component are specified by

1. First parameter is the component name
2. Rests of the parameters are the names of the input ports whose data values should satisfy causal consistency constraints. e.g. In Fig 2.9 Ports P4 and P5 of component C should satisfy causal consistency constraints

2.7 *atomic.txt* - Atomic Consistency Constraints

If a component is a composite component, the designer may specify atomic execution of the component. This implies that for any execution, there exists an equivalent execution in which all actions executed in response to an event arriving at a uses port are atomic with respect to the rest of the system. Names of such composite components can be specified on different lines in *atomic.txt* as shown in Fig. 2.10

Figure 2.10 *atomic.txt* file with Atomic Consistency Constraints

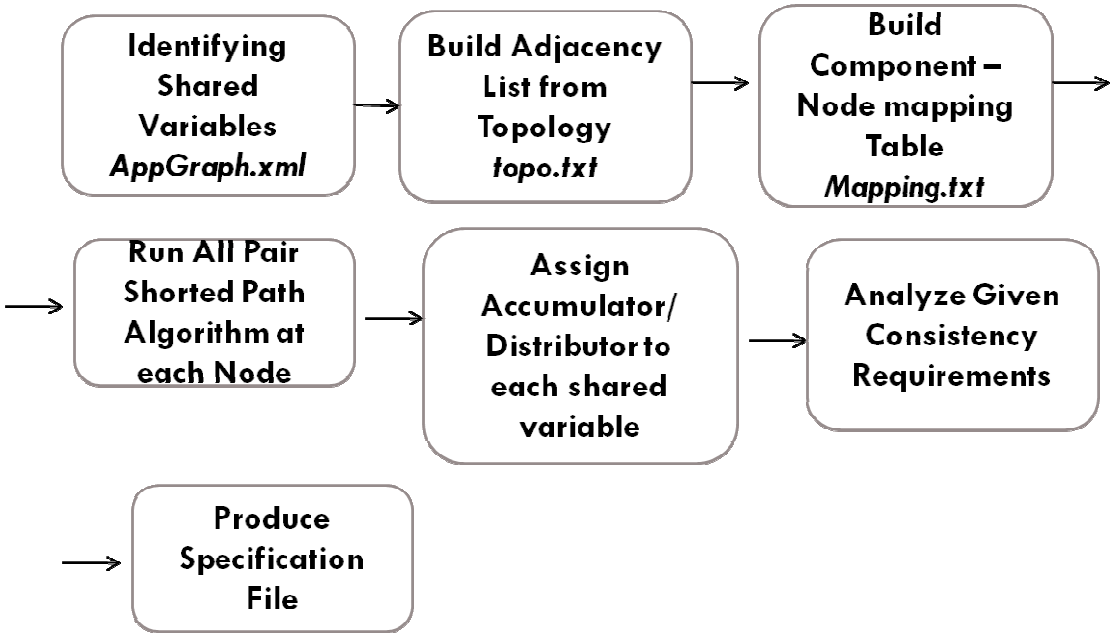
D

This specifies that the execution of operations of the nested components within component D should be atomic with respect to rest of the system.

CHAPTER 3 - Labeling Algorithm

The Labeling algorithm is one of the most important parts of the framework. We perform graph analysis to come up with labeling for each node which then is used by the nesC infrastructure during execution. Fig 3.1 shows various phases of a Labeling algorithm.

Figure 3.1 Examples of variable sharing



The first task of the Labeling algorithm is to use the file *App_Graph.xml,* described in Chapter 2, as an input and identify the shared variables present in the application graph. . Next, it reads the file *topo.txt* file to build up an adjacency list representing topology graph information given in the file. We then extract the mapping information of application components to nodes in the topology graph from *mapping.txt* file and link it with the topology graph adjacency list. All these data structures along with the given consistency requirements are then analyzed by the Labeling algorithm to come up with the labels for each shared variable to generate e the file specs.txt.

This chapter aims to describe in detail the mechanism used to identify the shared variables and then how labeling framework indentifies accumulators and distributors nodes in the

topology graph and message processing type based on the given consistency requirements.

3.1 Identifying Shared Variables

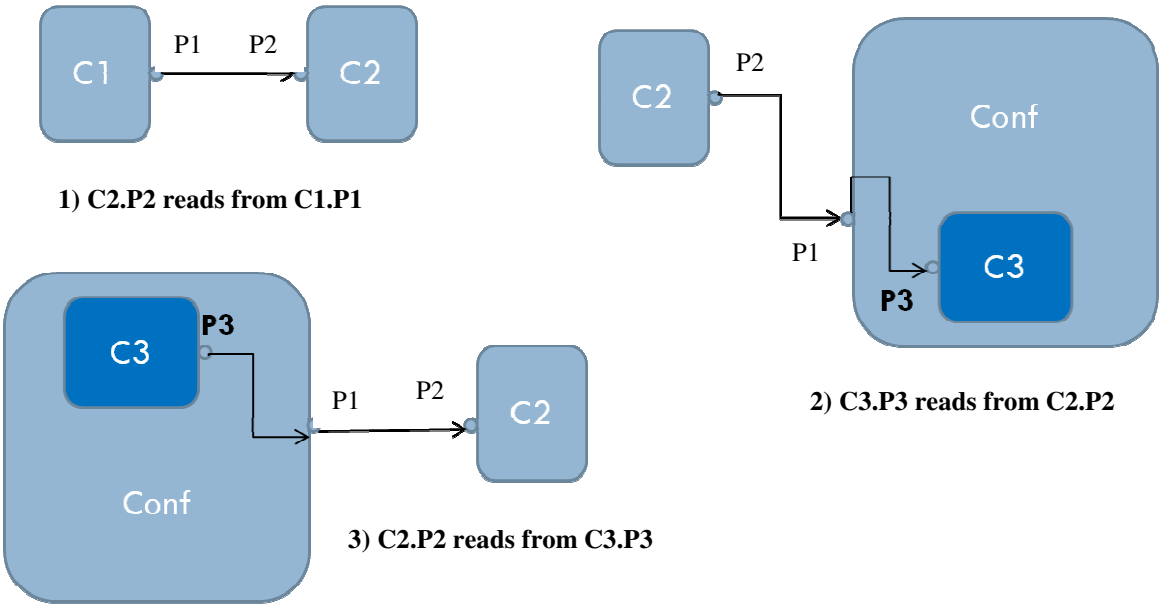
We now define how the variables are shared between the components. We define a mapping, reads_from, from input ports to output ports. As shown in Fig 3.1(1) if the output port p1 of C1 is wired to the input port p2 of C2, then we say that C2.p2 reads_from C1.p1. If input port p1 of a configuration Conf is mapped to port p2 of C2, then p2 is mapped to all ports which Conf.p1 is mapped to. Thus, if Conf.p1 reads_from C3.p3 then C2.p2 Reads_from C3.p3, as shown in Fig 3.1(2). Let output port p1 of a configuration Conf be mapped to port p2 of C2. If C3.p3 reads_from Conf.p1 then C3.p3 reads_from C2.p2, shown in Fig 3.1(3)

If a set, P1, of ports read from a set, P2, of ports, then we introduce a new variable, x, with all ports in P2 as writers of x, and all ports in P1 as readers of x. Note that each port in P1 must read from all ports in P2 and not any other additional ports. If P1 is a singleton set, then x is a single-reader, multiple-writer variable. If P2 is a singleton set, then x is a multiple-reader, single-writer variable. We will use x_c to denote the port on C which reads or writes x.

We have three cases to consider:

1. Multiple-writer, single-reader variable
2. Single-writer, multiple-reader variable
3. Multiple-writer, multiple-reader variable

Figure 3.2 Examples of variable sharing

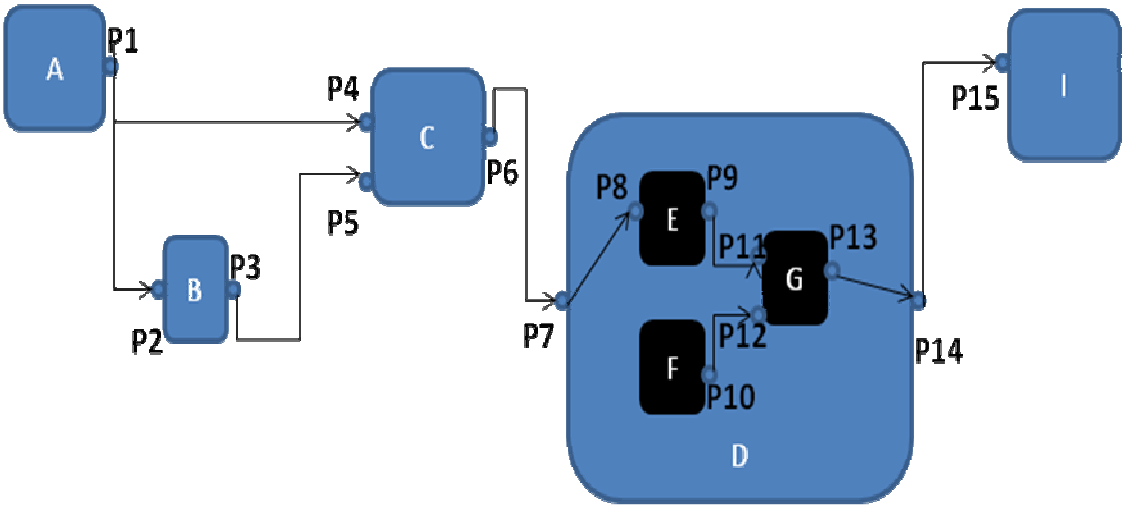


We analyze the application graph file to determine the variables identified above. We build the following tables from this analysis:

- Accumulator shared variable table: This table contains name of components involved in multiple writer single reader scenario
- Distributor shared variable table: This table contains name of components involved in multiple reader single writer scenario

Multiple-writer single-reader scenarios can be found out by looking at wiring information of input ports of all the components. If any input port is connected to more than one output port of other components then this is added to the Accumulator shared variable information table. On the similar lines, we can identify multiple-reader singlewriter scenarios by looking at output port wiring information of each port. If any output port is wired to more than one input port of other components then this will get added to the Distributor shared variable information table. After going through all the components and their wiring information, we are able to build the Shared Variable Information Tables for each type of shared variable we are trying to identify in a given application graph. This design gives us the flexibility of extending current implementation to support other types of shared variables which can be identified from the application graph in future.

Figure 3.3 Application Graph Example 1



Consider above scenario in Fig. 3.1 where output port (*provides*) P1 of component A is connected to the input ports (*uses*) P2 and P4 of components B and C respectively. Also, value read by component B on port P2 is written onto port P3. Values read by component C on input ports P4 and P5 should be causally consistent. Component D is composed of nested sub-components. Component D should satisfy atomic consistency. Port P7 of component D can be equated with input port P8 of component E also port P14 is equated with port P13 of component G. For atomic consistency one needs to delay delivery of message with sequence number *n* until messages with sequence number *n-1* are delivered to component G.

After analyzing this application graph as we can see that there is one Distributor shared variable present between components A, B and C. This information is stored in the Distributor information table as A -> B -> C i.e. the first node in the list will always be the node which is acting as a distributor followed by the list of all the reader components.

3.2 Building an adjacency list for the network topology

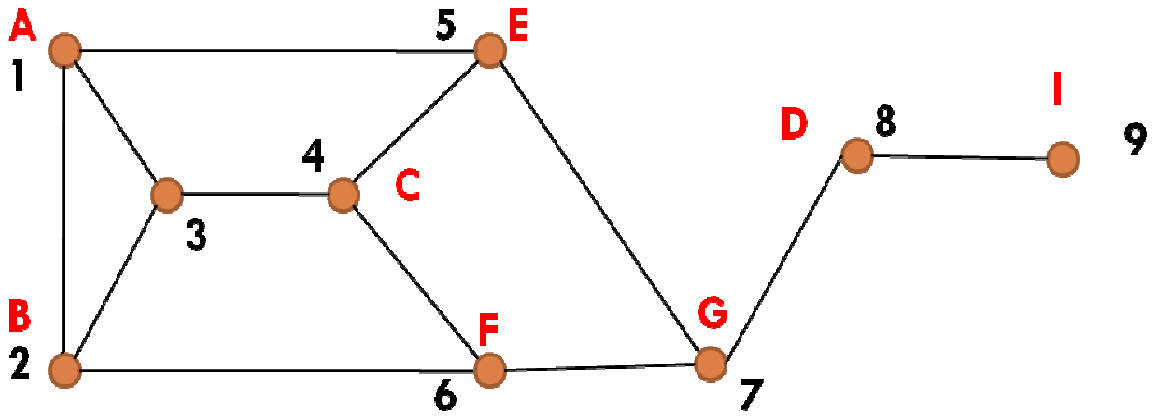
This phase reads the topology graph information given in the file *topo.txt* file and constructs an adjacency list representation of the physical topology. This adjacency list representation is then used to find shortest paths of each pair of nodes. This path information is then used to identify accumulator/distributor nodes. An accumulator node is one whose input port is connected to output ports of more than one different component. A distributor node is one whose output port is connected to input ports of more than one different component. An

accumulator node is identified for each multiple-writer, single-reader variable. Ideally, this variable must be a node whose distance to the writer nodes is the minimum. A distributor node is identified for each multiple-reader, single-writer variable. Ideally, this variable must be a node whose distance to the reader nodes is the minimum. This phase also reads the mapping information from *mapping.txt* file and maintains a table of mapping of each component on each node. We also provide an API which when given a component name returns the node number on which it is deployed which can be used at the time of labeling nodes.

```
get_mapping_information(component_name)
```

Consider following network graph where components are mapped onto different nodes as labeled in the Fig. 3.2

Figure 3.4 Network Graph Example 1



3.3 Identifying the Accumulator and Distributor Nodes

After we have all the data mentioned in Section 3.1 and Section 3.2, we start identifying accumulator and distributor nodes for each shared variable. This is done by finding the nearest common node between all the nodes on which the components involved in writing to a shared variables are mapped.

e.g. If we take an example mentioned in **Fig 3.1** where component involved in writing are mapped like this B →2; C→ 4

Then, we find the nearest common node between nodes 2 and 4 (from the topology graph adjacency list containing all pair shortest path information) which can distribute values to the reader components . The nearest common node obtained is the Distributor node for shared variable under consideration is represented as DIS (X1) where X1 is the shared variable associated with this reader-writer set. .

Similarly, for each accumulator shared variable, we find the node nearest to all the nodes on which the components involved in writing a shared variable are mapped.

Labeling for accumulator nodes is also done on the similar lines where only the assigned label is different. Similarly, we will go through all the shared variables identified in section 3.1 and after finding nearest common node we will label it as Acc(x) or Dis(x).

3.4 Analyzing Various Consistency Constraints

Various consistency constraints such as causality and atomicity can be read from the *causal.txt* and *atomic.txt*. For given Example 1 under consideration network graph in Fig 3.4, it can be seen that causal consistency can be violated at C. Consider a scenario where B receives *x* from A and then sends *y* to C. C also receives *x* directly from A. Here it is possible to receive value of *y* that is dependent on more recent value of *x*. For causal consistency, the Labeling algorithm reads different input ports information which require causality to be ensured and identifies paths along which causality can be violated. . E.g. If it is specified that ports P4 and P5 of a component C should satisfy causality, then the Labeling algorithm first identifies the source component A connected to port P4 on which values read by P4 and P5 are dependent on. After identifying the source component, it will try to find an alternative path from port P5 to component A and add labels for each node along this path to propagate sequence numbers. That is, a sequence number is generated at A and each node along this alternative path copies the incoming sequence number on to the outgoing message. This enables us to propagate dependency information along the path. .

The node with component C on it will have label indicating that the message delivery be delayed to ensure causal consistency for this node.

Atomic consistency can be specified for component D. Labeling algorithm first identifies all the nested components whose ports are equated with input port of component D. All nodes having these components will be labeled asking to copy sequence number of incoming message

on to each outgoing message. Other nested components such as G which read from more than one component will have label indicating that the message delivery be delayed to ensure delivery of messages with sequence number *n* until we have delivered messages with sequence number *n-1* on both input ports of G.

All this labeling information is then written into *specs.txt* which can then be used by NesC Consistency framework for further processing.

CHAPTER 4 - Consistency Framework

This is another important phase of this project which makes use of the *specs.txt* file generated from the Labeling framework mentioned in chapter 3. The purpose of specs.txt file is to build the shared variable processing tables at each node which can be used by Consistency framework to ensure data consistency. This will be done by reading the table information from the specification file

Before we look at how this framework makes use of the specs.txt file we will first go over the format of this file.

4.1 *specs.txt* File Format

Figure 4.1 illustrates the format of the sample specification file produced after applying the Labeling algorithm for the example discussed in Chapter 3.

Figure 4.1 Specification File structure Example 1

```
numnodes 9
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 4 -1
end
nodeid 2
1 3 -1
end
nodeid 4
2 2 -1
end
nodeid 5
2 3 -1
End
nodeid 6
2 2 -1
end
nodeid 7
2 5 -1
end
nodeid 8
2 1 -1
```

The following points will explain the node table parameters from Fig. 4.1 -

1) The number of nodes in the network specified by the keyword *numnodes*.
2) The node shared variable table information can be specified as follows -
   i. The node id under consideration is specified by keywork *nodeid* followed by the node number
   ii. The shared variable information begins with the shared variable number
   iii. Following that the first integer specifies the *processing_type* given shared variable belongs to i.e. 0 indicates shared variable belongs to forwarder message processing type, 1 indicates it belongs to accumulator message type, 2 indicate it belongs to distributor message type etc…
   iv. After that we may have list of parameters that one needs for processing of given *processing_type* terminated by -1 to indicate end of parameters
   v. After the information for all the shared variables have been specified, the end of the shared variable information must be indicated by keyword *end*

4.2 Consistency Framework Implementation

After looking at the structure we will now see how this file is actually used in the framework implementation. Simulation framework read the shared variable information associated and loads it into the internal table each node is maintaining only if, node id read from the *specs.txt* file is same as the TOS_LOCAL_ADDRESS. This means each node will just load the information associated with it and not other nodes.

After initializing service tables maintained by all the nodes (maintaining shared variable information) the job is to just intercept all the messages received from the communication layer *comm* and process them based on the processing information present in the service table by identifying processing type of the message by looking at message structure fields. We can have messages of various types like forwarder messages, accumulator messages etc… We will now look at how these messages are identified from *specs.txt*

Processing_type 1: Forward messages containing shared variable x to neighbor y

Processing _type 2: When message containing variable x arrive assign it a sequence number and forward

Processing _type 3: When messages containing variable x arrive store the sequence number in the message and when sending the message containing variable y, include this sequence number

Processing _type 4: When message containing variable x arrives store the variable locally. When a message containing y is being sent piggyback the value of x

Processing _type 5: Delay message delivery to ensure causal consistency

This processing type mechanism is really helpful as it can be easily extended to support messages of different processing types by just adding the processing code for newly added processing type. Any application which wants to ensure consistency can make use of the send and receive interfaces provided by given framework.

Now we will go over the specs.txt file in Fig 4.1. As specified from causal.txt file we know component C needs to ensure causal consistency. We will see how it will get satisfied by looking at only those labeling entries in the *specs.txt* file for all the nodes having entries for this particular identified shared variable.

```
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 -1
end
nodeid 2
1 3 -1
end
```

Node1 is acting as a distributor node for components B and C. So it has a label with *processing_type* 2 indicating it needs to assign sequence number to messages before sending them out. Node 2 is having 3 as a *processing_type* label for shared variable 1 indicating it needs to copy the incoming sequence number from data received from node 1 while it is sending the outgoing message. Similarly node 3 will act as a forwarder and it will forward the incoming data to node 4. Node 4 which has component C mapped onto it has a label with *processing_type* 5

indicating it needs to delay the delivery of messages with sequence number n until it receives messages with sequence number n-1. This will ensure that node 4 (component C) will process messages belonging to same time stamp ensuring causal consistency.

Now we will look at how atomic consistency will get ensured with the labels present in *specs.txt* file in Fig 4.1. Labeling framework identifies it as a shared variable 2. We will look at all the nodes having entries for shared variable 2.

```
nodeid 4
2 2 -1
end
nodeid 5
2 3 -1
nodeid 6
2 2 -1
endend
nodeid 7
2 5 -1
end
nodeid 8
2 1 -1
end
```

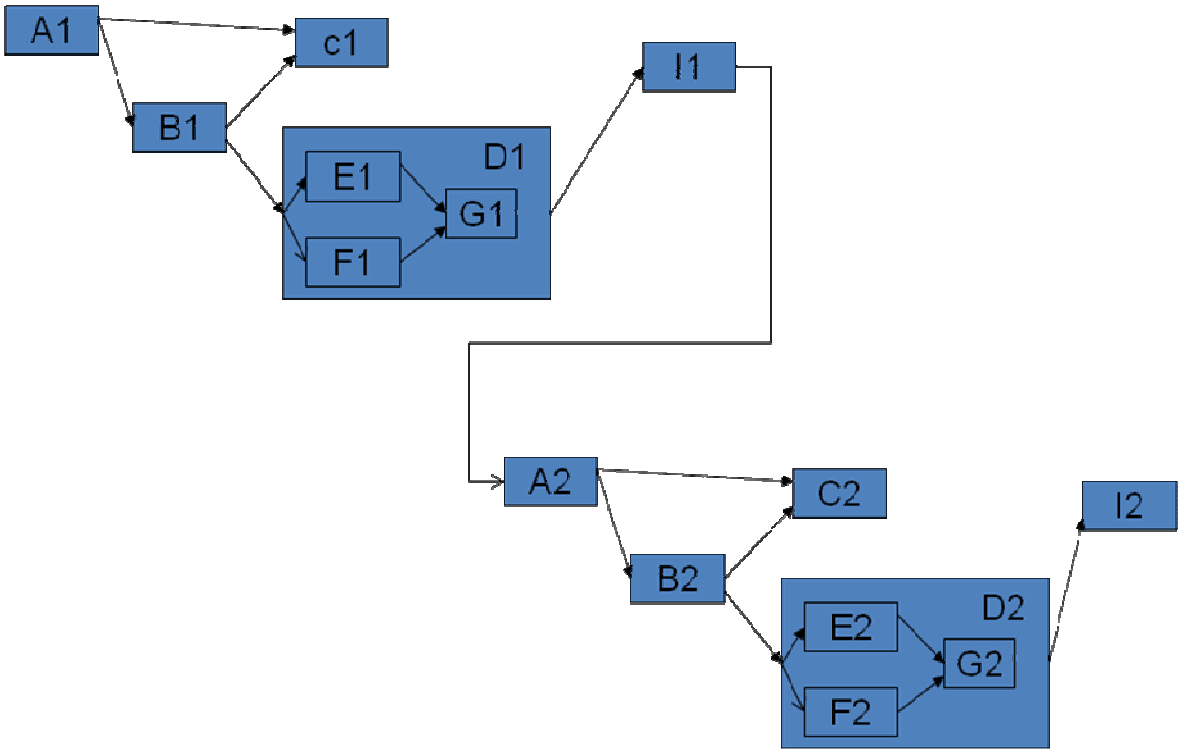
Node 4 and Node 6 will assign sequence number to the outgoing messages. Node 5 will copy the incoming sequence number while sending the outgoing message. Node 7 which has component G mapped onto it has a label with *processing_type* 5 indicating it needs to delay the delivery of messages with sequence number n until it receives messages with sequence number n-1. This will ensure that node 7 (component G) will process messages belonging to same time stamp ensuring atomic execution.

CHAPTER 5 - Performance Results

In this chapter we will analyze the overhead of our framework by looking at number of additional messages needed, amount of energy consumed etc… for various different scenarios.

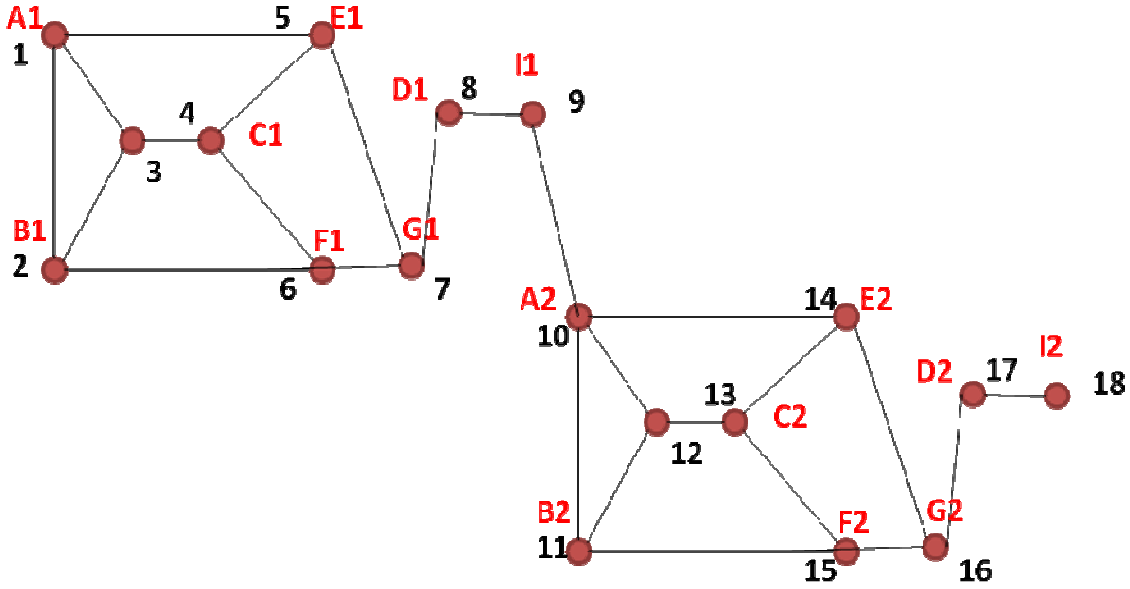
First we will analyze the impact of increasing number of components in a given scenario by keeping the ratio of number of components to number of constraints same as in example 1 in chapter 3. For this first we will consider an application graph shown in Fig 5.1 which is put together after serial replication of example 1 discussed in chapter 3. Similarly we have the network graph shown in Fig 5.2

Figure 5.1 Application Graph Scenario 1



In given application graph Components D1 and D2 needs to satisfy atomic consistency and components C1 and C2 needs to satisfy causal consistency.

Figure 5.2 Network Graph Scenario 1



After running labeling algorithm on given application graph *specs.txt* file shown in Fig 5.3 gets generated which will has labels for all the nodes in the network graph based on the consistency requirements mentioned earlier.

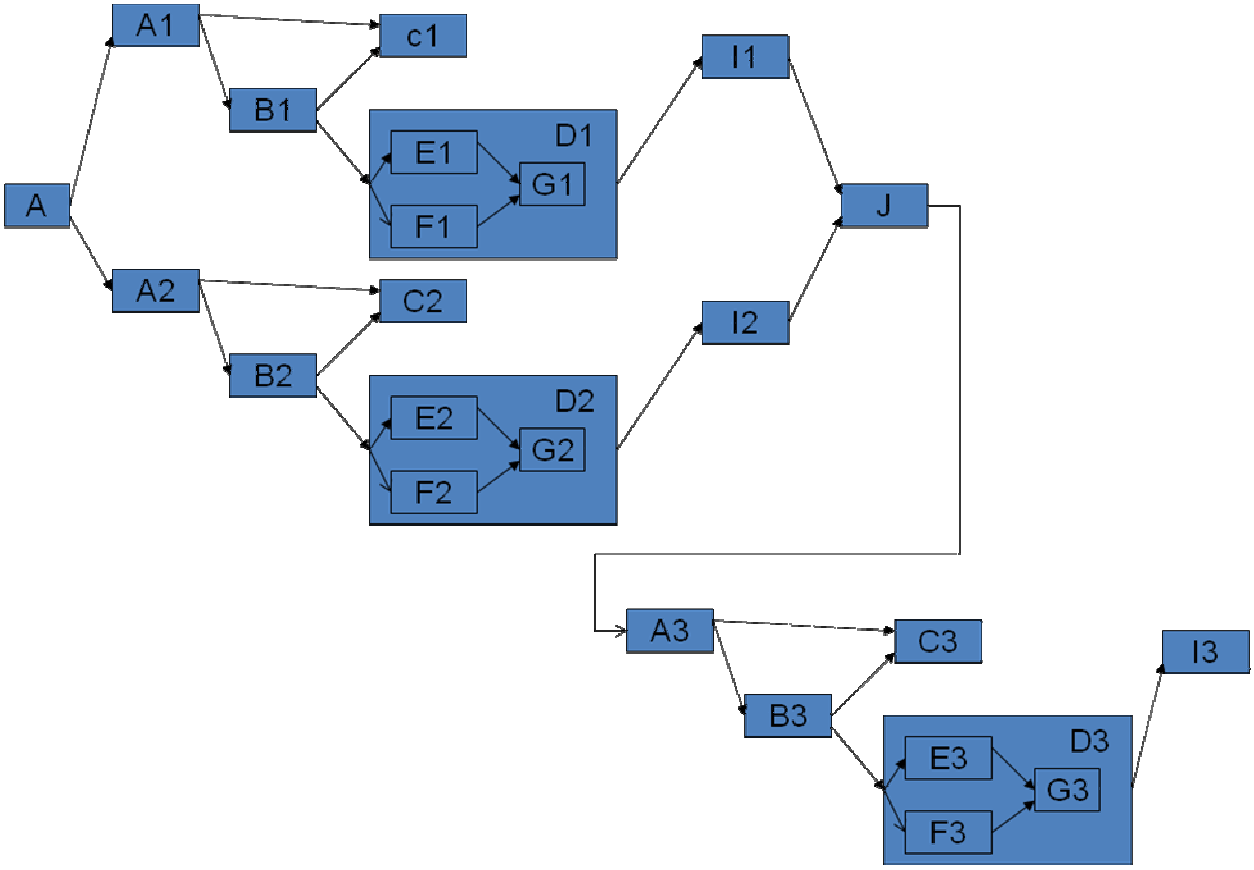
Figure 5.3 Generated *specs.txt* file for Scenario 1

```
numnodes 18
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 4 -1
end
nodeid 2
1 3 -1
end
nodeid 4
2 2 -1
end
nodeid 5
2 3 -1
End
nodeid 6
2 3 -1
end
nodeid 7
```

```
2 5 -1
end
nodeid 8
2 1 -1
nodeid 9
2 1 -1
nodeid 10
3 2 -1
end
nodeid 13
3 5 -1
end
nodeid 12
3 1 13 -1
end
nodeid 11
3 3 -1
end
nodeid 13
4 2 -1
end
nodeid 14
4 3 -1
End
nodeid 15
4 3 -1
end
nodeid 16
4 5 -1
end
nodeid 17
4 1 -1
nodeid 18
4 1 -1
end
```

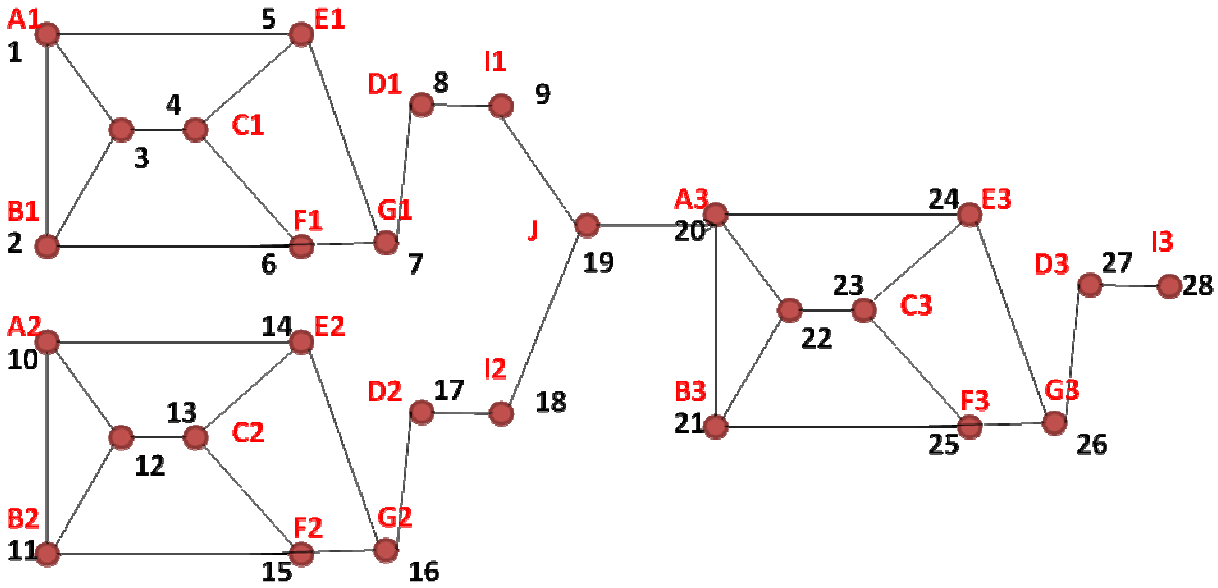
Now we will modify the scenario 1 by increasing the number of components. Consider following application graph in Fig 5.4 which is put together after parallel and serial replication of example 1 discussed in chapter 3. Similarly we have the network graph shown in Fig 5.5

Figure 5.4 Application Graph Scenario 2



In given application graph Components D1, D2 and D3 needs to satisfy atomic consistency and components C1, C2 and C3 needs to satisfy causal consistency. As we can see scenario 2 has more number of components compared to scenario 1 but constraint to component ratio is same for both the scenarios which will help us in analyzing impact of increase in number of components on the consistency framework.

Figure 5.5 Network Graph Scenario 2



After running labeling algorithm on given application graph *specs.txt* file shown in Fig 5.6 gets generated which will has labels for all the nodes in the network graph based on the consistency requirements mentioned earlier.

Figure 5.6 Generated *specs.txt* file for Scenario 2

```
numnodes 28
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 4 -1
end
nodeid 2
1 3 -1
end
nodeid 4
2 2 -1
end
nodeid 5
2 3 -1
End
nodeid 6
2 3 -1
end
nodeid 7
```

```
2 5 -1
end
nodeid 8
2 1 -1
nodeid 9
2 1 -1
nodeid 10
3 2 -1
end
nodeid 13
3 5 -1
end
nodeid 12
3 1 13 -1
end
nodeid 11
3 3 -1
end
nodeid 13
4 2 -1
end
nodeid 14
4 3 -1
End
nodeid 15
4 3 -1
end
nodeid 16
4 5 -1
end
nodeid 17
4 1 -1
nodeid 18
4 1 -1
nodeid 20
5 2 -1
end
nodeid 23
5 5 -1
end
nodeid 22
5 1 23 -1
end
nodeid 21
5 3 -1
end
nodeid 23
6 2 -1
end
nodeid 24
6 3 -1
End
nodeid 25
6 3 -1
end
nodeid 26
6 5 -1
```

```
end
nodeid 27
6 1 -1
end
```

Power consumed by each node is calculated by making use of the power profiling available in TOSSIM. It is calculated by subtracting the power available with each node at the end from the power available at each node in the beginning. Power available at each node can be seen by executing tinyos application with following command line options.

```
export DBG= power,usr1
./build/pc/main.exe -p 28
```

Figure 5.7 output for Scenario 2

```
****NODE1****
Number of messages recd = 0
Number of messages sent = 10
Power consumed = 110 mA
****NODE2****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 110 mA
****NODE3****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 224 mA
****NODE4****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 220 mA
****NODE5****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 98 mA
****NODE5****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 97 mA
****NODE7****
Number of messages recd = 5
Number of messages sent = 10
Power consumed = 164 mA
****NODE8****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 114 mA
****NODE9****
Number of messages recd = 0
Number of messages sent = 5
Power consumed = 57 mA
```

```
****NODE10****
Number of messages recd = 0
Number of messages sent = 10
Power consumed = 110 mA
****NODE11****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 118 mA
****NODE12****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 235 mA
****NODE13****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 280 mA
****NODE14****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 130 mA
****NODE15****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 108 mA
****NODE16****
Number of messages recd = 5
Number of messages sent = 10
Power consumed = 178 mA
****NODE17****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 114 mA
****NODE18****
Number of messages recd = 0
Number of messages sent = 5
Power consumed = 65 mA
****NODE19****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 210 mA
****NODE20****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 234 mA
****NODE21****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 110 mA
****NODE22****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 224 mA
****NODE23****
Number of messages recd = 10
Number of messages sent = 10
Power consumed = 230 mA
****NODE24****
Number of messages recd = 5
```

```
Number of messages sent = 5
Power consumed = 118 mA
****NODE5****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 125 mA
****NODE26****
Number of messages recd = 10
Number of messages sent = 5
Power consumed = 187 mA
****NODE27****
Number of messages recd = 5
Number of messages sent = 5
Power consumed = 113 mA
****NODE28****
Number of messages recd = 5
Number of messages sent = 0
Power consumed = 68 mA
```

We can see that after increasing number of components in the application graph only the labeling information maintained at each node increases which will increase the time required to process each message by those nodes. This can be seen by looking at the generated output shown in Fig 5.8 comparing performance for Scenario1 and Scenario2, we can see some more nodes got added in Scenario 2 as we increased number of components for keeping information about added components.

Following is the output showing tabular representation of performance results obtained for 2 different scenarios under consideration with different number of components. Each entry in scenario column represents a tuple of the form (Messages sent, Messages recd, Power Consumed, Total number of Bytes in Messages sent).

Figure 5.8 Tabular representation of Performance Results 1

Node Id	Scenario 2	Scenario 2
1	(0,10,110,100)	(0,10,110,100)
2	(5,5,110,100)	(5,5,110,100)
3	(10,10,225,200)	(10,10,225,200)
4	(10,10,220,200)	(10,10,220,200)
5	(5,5,120,100)	(5,5,120,100)
6	(5,0,72,50)	(5,0,72,50)
7	(5,10,180,150)	(5,10,180,150)
8	(5,5,114, 100)	(5,5,114, 100)

9	(5,5,120,100)	(5,5,120,100)
10	(0,10,120,100)	(0,10,120,100)
11	(5,5,118,100)	(5,5,118,100)
12	(10,10,235,200)	(10,10,235,200)
13	(10,10,280,200)	(10,10,280,200)
14	(5,5,130,100)	(5,5,130,100)
15	(5,5,108,100)	(5,5,108,100)
16	(5,10,178,150)	(5,10,178,150)
17	(5,5,126,100)	(5,5,126,100)
18	(5,0,68,50)	(5,5,120,100)
19	–	(10,10,210,200)
20	–	(10,10,234,200)
21	–	(5,5,110,100)
22	–	(10,10,234,200)
23	–	(10,10,230,200)
24	–	(5,5,107,100)
25	–	(0,5,53,50)
26	–	(10,5,169,150)
27	–	(5,5,110,100)
28	–	(5,0,68,50)

After analyzing the number of messages send and received by making use of consistency framework it can be seen that this framework puts very minimal overload on the overall performance of the application as count of number of messages send and received by each node remains the same. Also as the ratio of constraints to components remains the same for both the scenarios the total number of bytes send in messages also remain the same. Scenarios 2 maintains some extra information as it has more number of components than Scenario .1 The nodes having various consistency constraints need to maintain some extra data structure and perform some computation to satisfy consistency and hence they consume very little extra power and memory space. But it can be seen that we need to add very minimal amount of information in order to ensure consistency.

Now we will analyze the impact of keeping the number of components in a given application graph in Scenario 2 the same but varying the component to constraint ratio to analyze a change in the overhead. We will carry out experiments by increasing number of constraints. Scenario3 specifying consistency constraint only for Component C1, Scenario4 specifying for

components C1, D1, C2 and Scenario2 we have already got the results by specifying constraints for components C1,D1, C2,D2, C3,D3.

If we consider Scenario3 just specifying consistency constraint for component C1 then labeling algorithm generates labels for only those nodes involved in ensuring consistency at C1. Fig 5.9 shows generated *specs.txt* file.

Figure 5.9 *specs.txt* file generated for Scenario 3

```
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 4 -1
end
nodeid 2
1 3 -1
end
```

As we have less number of entries present in specs.txt file, only those nodes having label's present in *specs.txt* file will initialize the shared variable information table with labeling information. So at the time of execution only those nodes having labeling information present in their table will process messages to ensure consistency other nodes will process them as any other message without caring about the consistency requirements reducing the amount of overhead involved in processing each message based on the labeling information. The number of messages exchanged between nodes is not dependent upon the consistency constraints. Consistency constraints add up some extra processing that needs to be done while sending or receiving any message to ensure consistency consuming little extra power needed for processing.

Similarly we can see that even after adding consistency constraints for components D1, C2 only the labeling information maintained at each node increases which will increase the time required to process each message by those nodes. This can be seen by looking at the generated output shown in Fig 5.11 comparing performance for Scenario2, Scenario3 and Scenario4 where we can see some decrease in the power consumed for nodes not maintaining any table for processing messages to ensure consistency under column for Scenario4. All nodes after node number 13 don't have any labels generated for them and consume little less power compared to

the output in Fig 5.7 and the specification file looks similar to the file in Fig 5.6 but containing labels for nodes up to 13 only as shown in Fig 5.10

Figure 5.10 Generated *specs.txt* file for Scenario 4

```
numnodes 28
nodeid 1
1 2 -1
end
nodeid 4
1 5 -1
end
nodeid 3
1 1 4 -1
end
nodeid 2
1 3 -1
end
nodeid 4
2 2 -1
end
nodeid 5
2 3 -1
End
nodeid 6
2 2 -1
end
nodeid 7
2 5 -1
end
nodeid 8
2 1 -1
nodeid 9
2 1 -1
nodeid 10
3 2 -1
end
nodeid 13
3 5 -1
end
nodeid 12
3 1 13 -1
end
nodeid 11
3 3 -1
end
nodeid 13
4 2 -1
end
```

Following is the output showing tabular representation of performance results obtained for 3 different scenarios under consideration with different constraints to components ratio. Each

entry in scenario column represents a tuple of the form (Messages sent, Messages recd, Power Consumed, Total number of Bytes in Messages sent). We add an entry for Total number of Bytes in Messages sent because in some cases number of messages sent may be the same but number of bytes may be different. Each message is composed of sequence number, shared variable number, processing type and data fields. If there are no consistency constraints then the nodes not having any label's will not make use of share variable number and processing type fields present in the message structure reducing on the total number of bytes sent in each message.

Figure 5.11 Tabular representation of Performance Results

Node Id	Scenario 2	Scenario 3	Scenario 4
1	(0,10,110,100)	(0,10,107,100)	(0,10,110,100)
2	(5,5,110,100)	(5,5,110,100)	(5,5,110,100)
3	(10,10,225,200)	(10,10,220,200)	(10,10,224,200)
4	(10,10,220,200)	(10,10,220,200)	(10,10,220,200)
5	(5,5,120,100)	(5,5,98,60)	(5,5,123,100)
6	(5,0,72,50)	(5,0,62,30)	(5,0,69,100)
7	(5,10,180,150)	(5,10,161,90)	(5,10,174,100)
8	(5,5,114, 100)	(5,5,97,60)	(5,5,114,100)
9	(5,5,120,100)	(5,5,102,60)	(5,5,124,100)
10	(0,10,120,100)	(0,10,108,60)	(0,10,122,100)
11	(5,5,118,100)	(5,5,108,60)	(5,5,118,100)
12	(10,10,235,200)	(10,10,198,120)	(10,10,235,200)
13	(10,10,280,200)	(10,10,210,120)	(10,10,268,200)
14	(5,5,130,100)	(5,5,110,60)	(5,5,110,60)
15	(5,5,108,100)	(5,5,96,60)	(5,5,96,60)
16	(5,10,178,150)	(5,10,160,90)	(5,10,160,90)
17	(5,5,126,100)	(5,5,102,60)	(5,5,102,60)
18	(5,5,120,100)	(5,5,104,60)	(5,5,104,60)
19	(10,10,210,200)	(10,10,198,120)	(10,10,198,120)
20	(10,10,234,200)	(10,10,228,120)	(10,10,228,120)
21	(5,5,110,100)	(5,5,104,60)	(5,5,104,60)
22	(10,10,234,200)	(10,10,224,120)	(10,10,224,120)
23	(10,10,230,200)	(10,10,229,120)	(10,10,229,120)
24	(5,5,107,100)	(5,5,107,60)	(5,5,107,60)
25	(0,5,53,50)	(5,5,53,60)	(5,5,53,60)
26	(10,5,169,150)	(10,5,169,90)	(10,5,169,90)
27	(5,5,110,100)	(5,5,110,60)	(5,5,110,60)
28	(5,0,68,50)	(5,0,68,30)	(5,0,68,30)

As we can see from above table, as the constraints to component ratio increases the amount of power consumed along with the number of bytes sent in each message increases.

References

1. TinyOS website http://www.tinyos.net/

2. Kewei Sha and Weisong Shi: Modeling Data Consistency in Wireless Sensor Networks

3. Vijaykumar Krishnaswamy, Michel Raynal and David Bakken: Shared State Consistency for Time-Sensitive Distributed Applications

4. Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan: Adaptive Protocols for Information Dissemination in Wireless Sensor Networks