Cougar Design Document

Johannes Gehrke, Niki Trigoni, and Yong Yao
{johannes, niki, yao}@cs.cornell.edu
Jay Ayres, Nick Gerner, Joel Ossher, and Lin Zhu
{kja9, nsg7, jpo5, lz26}@cornell.edu

Version 0.1

Contents

Abstract

The importance of coordinated, energy efficient, hetrogenous network query processing increases as distributed, embedded and possibly mobile devices proliferate. The time will come when many devices of wildly different types and capabilities will be asked to coordinate and deliver data that they carry and collect to one or several end users. The Cougar project seeks to identify and address the issues in providing this capability using existing and future hardware/software.

This document is meant as an explaination of the work completed so far and the direction of work yet to come. Architecture, modules, interfaces and implementations already in place as well as specific future coding plans are discussed.

I have also detailed my contributions to the project up to this point.

1 Introduction

The research goal of the Cougar project is to explore in network query processing for a sensor network. The approach taken is to view the network as a distributed database. Each node can generate tuples in response to queries posed to the network. We have chosen to investigate a hetrogenous network consisting of many nodes each possibly having different capabilities including network connectivity. Some nodes may be capable of long range or high speed network communication, while others may only be able to communicate at very low bandwidth to neighbors within several meters. Different nodes may have different sensor capabilities. Some nodes may be able to generate light, temperature and sound readings, while others may only generate one or two of these, or separate readings altogether.

Over the last semester my primary contribution to the Cougar project has been to write implementations of some of the research ideas generated in weekly research meetings and independently by Professor Gehrke, Nikki Trigoni and Yong Yao. In addition to these weekly research meetings we also met weekly to discuss research papers relating to our work. I have listed some of these papers that have been useful at the end of this paper.

The implementing team, I, Jay Ayers, Joel Osser and Lin Zhu, met throughout every week. Coordination has been, perhaps, the most difficult and important part of the work we have done. Throughout the last semester I have tried to concretely identify our immediate goals (goals defined abstractly in our weekly research meetings with Professor Gehrke, Nikki and Yong). Once goals are defined it becomes important to identify modules that can be independently developed so the four of us on the implementing team can work separately throughout the week to meet our design goals. I cannot stress enough the importance of modular coding. Dividing the code into modules allows independent development over a short period of time (a week or two). In this time functionality can be implemented and the code can be tested. Even before a module is complete, other, dependent, modules can be written using abstract interfaces which we agree upon before we begin coding. This effort allows us to develop separately and then join our works together in exactly the way that all of us expect. This alsoreduces ambiguities about what each of us has written and

how what each of us has written works. This is what I have tried to focus on in the last semester.

I have also put together nightly tests that check out the latest code from a central server, compiles it, and runs tests against known results. This allows us to be confident that not only the modules we have written separately, but all of the code together performs as expected. As progress continues these nightly tests will also allow us to conduct time intensive experiments and automatically collect data so we can spend our time analyzing that data and refining our implementations.

2 Methods and Materials

In developing methods for in-network query processing over a sensor network we have designed and begun implementing a network stack architecture for use in a query processor. One of the most important features of our network stack design is its portability. From the start we have kept in mind that we don't have a a system platform for which we're designing. Instead we have endevored to make what we write as general and portable as possible. To this end the lowest layer of our network stack is an adapter class that allows the rest of the stack to interface with any network communication facility. This allows us to build our query processor on a standard linux or windows based PC using TCP or UDP networking to simulate single hop communication. And this is exactly what we've done up to this point. But all of the code above this adapter class relies on a simple communication interface which could be implemented on a variety of platforms and network infrastructures.

The network stack we've developed consists of, at the lowest layer, what we've called the Link Layer. This layer is responsible for carrying out the lowest level of network communication. Nodes can broadcast messages to locally connected neighbors. On top of this layer we are currently building a dispatch layer which is closely integrated with the routing layer. Together these two layers are responsible for single hop, unicast, and multihop communication. The structure of the routing layer is probably the most meaningful contribution to the project so far. We're using semantic, attribute-value pairs to identify nodes and messages. I'll discuss these ideas further below.

We've also developed a testing architecture which we can use to run applications on our network stack as we develop it. This architecture coordinates different nodes to build the node and network structure, run commands on nodes and maintain a network of hetrogenous nodes. All of these features are useful during initial development, but will be invaluable once we begin serious experimentation. I'll outline the testing architecture below.

2.1 Link Layer

The Link Layer is really an interface to the lower level networking that the underlying platform supports. The interface provided by the Link Layer is simple, single hop, broadcast. We support multiple network interfaces since

we have assumed a hetrogenous network of nodes of perhaps multiple, varying network communication capabilities. Upper layers specify the network interface and the data to transmit. These upper layers may also provide a call-back for message reception. Because of this call-back message reception it is important that the upper layers be aware of blocking processing. There is no guarantee that the Link Layer implementation will handle blocking call-backs in a network traffic safe way. That is to say, network traffic may be missed if the upper layer call-back blocks.

Currently we have written two Link Layer implementations so far, a single threaded simulator and a distributed simulator.

2.1.1 Single Threaded Link Layer

The first implementation we wrote is a single threaded network simulator. This simulator provides all of the above functionality in a single thread. Many Link Layer sits on top of a "Physical Layer" which provides a single network queue. When a message is sent by an application, the Link Layer passes it on to the Physical Layer along with it's node id. The Physical Layer then adds the message to a single network queue and then dequeues all messages from the queue dispatching messages to the correct Link Layer via a call back. Upon reception of a message from the Physical Layer, the Link Layer executes all application call backs with the message data. Since the entire network runs in a single thread, all nodes must be run within the same program, although upper layers may run in multiple threads.

2.1.2 Distributed Simulator Link Layer

The second implementation we wrote is a multiple threaded, distributed network simulator. This simulator uses UDP to send messages from one node to another. Currently there is a central server to which all nodes direct their messages. This central server then identifies neighboring nodes and sends the message on to those nodes. On reception of a message the Link Layers, which may reside on separate computers execute application call backs.

It should be clear that both implementations provide the same functionality as far as the upper layer application code is concerned. Both provide the same Link Layer interface described above. Both send and recieve messages to and from neighboring nodes. Both communicate with upper layer application using the same call back functionality.

One feature which I have not discussed thus far is how neighboring nodes are identified in either simulator. To accomplish this we have developed an XML configuration file. This file specifies various node configuration information including, most notably, network connectivity.

2.2 Dispatch and Routing Layer

Currently we are developing this layer of the network. I believe this layer to be the most important and interesting work we've done so far. First I'll describe the attribute-value message structure and routing. Then I'll go on to discuss our plans for routing. As I noted above we are still working on this area of our research so there are a lot of incomplete ideas here that still need to be fleshed out. I will try to identify what we have implemented already and plan to keep and discuss these ideas here. There are many ideas we are still considering and I will discuss these in a later section.

The Dispatch and Routing Layer provides methods which upper layers may use to establish and interest in network traffic. These interests indicate that network traffic matching the upper layer's pattern, using an attribute-value list, should be directed to that node. The Dispatcher should notify the upper layer that such traffic has arrived when it does. Application code may also snoop traffic matching a pattern that incidentally arrives at the node. The difference between these two methods is that the first is meant to indicate that the node is a multihop destination of matching network traffic. This means that nodes must coordinate their routing tables when some interests are established by upper layers. In this way nodes are addressed by the attribute-value interests that application code registers with the Dispatch and Routing layer.

Already we have developed a Tree Manager that organizes routing trees. Since query evaluation uses trees rooted at the node that originates the query a tree manager of this form is very important for query processing. When a node poses a query to the network, the routing layer first checks the query against all existing trees to see if a suitable tree, or several trees that together will be suitable, already exists. If so, no new tree need be created and the query can be passed down the tree to participating nodes. If not the query is flooded through the network and a query tree is constructed. Children nodes then respond to the query and pass results back up the tree to the root node.

Thus far we have no complete implementation of the Dispatch and Routing Layer. In fact we are still discussing interfaces and functionality that the Dispatch and Routing Layer should provide. Query processing will have to be closely integrated with this layer since in network query processing is a research goal for the Cougar project. So complete work on this layer will not be ready until we have progressed with query evaluation.

2.3 Testing Architecture

There were several goals in the creation of our testing architecture. Among others was that we be able to run any implementation of the network stack with any node applications we might develop. Also the architecture should be flexible so that as we developed the network stack, from the bottom up we would be able to expand our tests and experiments without compromising previous work and without forcing the creation of a new testing structure with every step of development.

The ability to run any network implementation with any node application is critical since from the start we had in mind a hetrogenous network of sensor nodes each with different capabilities and potentially many implementations of all the network layers. On top of each node's network stack we have plan to have different applications. All of this means careful use of interfaces and implementing classes. At every stage we started with an interface that would provide uniform access to module functionality across implementations.

Since the research involves query processing at several layers of network communication it is important that we be able to manipulate layer functionality across network layers. Of course we still need the network stack and query processing to be self contained. So we came up with a set of interfaces and implementing classes that allow us to experiment with any module independently. Since our network stack is being built bottom up this means that every layer can be used as a foundation with testing code sitting on top of it without concerning the testing code with lower or upper layers. This approach gives us the ability to test as we develop and apply ideas across the network stack even after upper layers are in development (hopefully without changes requiring too many corresponding changes to the upper layers).

To accomplish all of this we have two executable classes that act as testing harnesses for node applications: GlobalTester and LocalTester. The first is a coordination center. The second runs node applications. Node applications are implemented by classes extending an abstract class MiniApp. We also wrote a configuration file format that allows us to specify network connectivity and node properties. This file is important for initializing simulated networks and coordinating some node properties across the network.

2.3.1 GlobalTester

GlobalTester is responsible for coordinating network tests. It reads the network configuration file and testing scripts. In the case of the Distributed Link Layer the GlobalTester starts the network communication server which receives all network traffic and forwards it to the correct nodes. In the case of the Single Threaded Link Layer the GlobalTester runs the single Physical Layer and all node applications, in much the same way as LocalTester runs node applications described below.

2.3.2 LocalTester

LocalTester starts node applications, initializing them from the configuration file. It also reads test scripts and waits for the GlobalTester to begin the network test. LocalTester starts the Distributed Link Layer client and connects to the GlobalTester's server. Once the network test begins test script commands are fed to node applications which respond by performing whatever actions they have been programmed with and logging output.

2.3.3 MiniApp

MiniApp is an abstract class that can be extended to perform node application functionality. MiniApp extensions fill the gap between the external node interface and the portion of the network stack of interest. In this way network code can be tested and experimented on as it is developed. Using this architecture we need not wait for the entire network stack to be complete before we begin developing network integrated query processing.

3 Discussion

Since this project is still in the early stages of development, we have no experimental results to speak of. We have been running nightly tests to verify the correctness of what we have developed so far. These tests test functionality against known results. Since we have only completed the Link Layer at this point we have not been able to begin experimenting with any of the research ideas we've developed at this point. In the coming weeks we will have the opportunity to begin applying some of the ideas we've been discussing. I will discuss some of these ideas below.

We should be able to implement a variety of query concious routing algorithms using the network stack we're developing. What's more is any code we write we should be able to install across platforms using the modular structure I've described above.

One of the first routing algorithms we could implement in this way is to have nodes locally pose interests for data with their neighbors. A node initially posing a query to the network will register an interest with its neighbors which will recursively do the same. Once results begin streaming back to the sink node which started the query routing selection can begin to minimize communication costs and still deliver the highest quality data to the sink. This query processing/routing method is detailed in Directed Diffusion[1].

Another query processing/routing method we could implement is a sleep/wake cycle routing algorithm. The specifics of the algorithm allow for in network aggregation of data for many different aggregation types. This method is detailed in TAG[2].

We also plan to develop semantic tree routing which I began to discuss in the section on the Dispatch and Routing Layer. Nodes register interests in certain kinds of network traffic based on the network message content. In this way nodes are addressed by the attribute-value interests that have been registered and any relevant network traffic will be directed to these nodes. This method poses a variety of problems to investigate including tree construction and maintenance, sleep/wake cycles and network traffic merging for duplicate traffic and splitting for multiple sinks.

4 Conclusion

In this document I've tried to indicate the overall structure we've laid out so far, which parts of that structure we've worked on already and other directions we plan to investigate. It's important to keep in mind that the Cougar project as far as the implementation we've been working on for the last year is still in the early stages of development. What we have tried to do in the last year is to develop some road map and direction to begin moving in. The research goals of the Cougar project have been laid down and work is progressing.

References

[1] Chalermek Intanagonwiwat, Ramesh Govindan, Debora Estrin "Directed Diffusion: A Scalable Robust Communication Paradigm for Sensor Networks", in *Proceedings of the Sixth Annual International Conference on Mobile Computing and Networking (MobiCOM '00), August 2000*, Boston, Massachussetts

[2] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong "TAG: a Tiny AGregation Service for Ad-Hoc Sensor Networks" in *5th Annual Symposium on Operating Systems Design and Implementation, December, 2002*

[3] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong "Aquisitional Query Processing" *SIGMOD, June 2003*, San Diego, CA.

[4] Wei Ye, John Heidermann, Deborah Estrin "Medium Access Control with Coordinated, Adaptive Sleeping for Wireless Sensor Networks" *USC/ISI Technical Report ISI-TR-567, January 2003*

[5] Nikki Trigoni, Yong Yao, Alan Demers, Johannes Gehrke, Rajmohan Rajaraman "Wave Scheduling"

[6] Praveen Seshadri http://www.cs.cornell.edu/Info/Projects/PREDATOR/designdoc.html *Predator: Design and Implementation*