Peer-to-Peer Data Sharing Among LabVIEW Nodes

Sadia Malik

CS 395T- Grid Computing
Project Report

12/13/03

INTRODUCTION

The goal of this project is to impose a peer-to-peer (P2P) data grid on a set of LabVIEW (LV) nodes. LV nodes, in this context, are data acquisition sensors that access live data. This data is published over the network via a publish-subscribe protocol used by these sensors, and any client who knows the physical address of these sensors can directly get the sensor data from the node that owns the sensor. These nodes are also capable of running embedded LabVIEW Real-Time (LVRT) applications, which allows us to create a P2P network with these nodes by creating a global namespace for these sensors [1] such that a client can use a logical name of the sensor and query the P2P grid to locate the sensor.

This report is organized as follows: First, I give a brief background of P2P computing model, sensor networks, and relevant applications [1]. Second, I describe the experimental setup and hardware I used to create the P2P netwok. The third section gives a detailed description of the P2P network design and the LV based implementation. The final section gives expected results and conclusion.

BACKGROUND

This project is based on two broad research areas in grid computing: P2P networks and sensor networks [1]. Both of these technologies have been around for some time [2], but the broad use of Internet technology has created several new opportunities for these technologies. Because these technologies are still being defined, I give a brief overview of each as it applies to this project.

P2P Computing Model

The P2P computing model allows different nodes on the network to communicate directly with each other to share data and/or resources. For this project, P2P is defined as follows [3]:

- Nodes have awareness of other nodes
- Peers create virtual network that abstracts the complexity of interconnecting peers
- Each node can act as both a client and a server
- Peers form working communities of data and application
- Overall performance of the system increases as more nodes are added

Sensor Network

Sensor network covers the distribution of small and relatively cheap sensors that are capable of wireless communication and significant computation [4]. Sensor network technology allows us to place sensors closer to the phenomena being monitored so that the data can be processed and filtered closer to the data source. It also allows multiple sensors to collect the same information and then collaborate with neighboring nodes to resolve ambiguities. Typical sensors in this network have embedded processor, wireless communication circuitry, data storage capacity, etc.

The sensor network technology is revolutionary for many situations where information gathering and processing is needed for inhospitable physical environments and/or less accessible environments, such as remote geographic regions or large industrial plants [4].

Application

This project will be a prototype of a real life application that can benefit from a P2P system which allows sharing of data in a standard data grid format. The real life application can be a subset of emergency control response system to a natural disaster, for example, earthquake or a tornado. In a situation like that, data needs to be acquired from different, spatially distributed sensors, and analyzed for different behaviors; for example, to predict the next course for the disaster and damage caused by it. Currently most of these systems use central data storage and supercomputers to store and process this data [5].

As distributed and networking technology becomes faster and more reliable, more and more applications are moving away from central repository for live sensor data. In this project, I'll present a solution that stores the data locally, in a distributed data grid, by combining the storage devices on all sensors and sharing that storage so that different sensors can put their data in a virtual global data space. This not only allows sensors to share unused storage space, but by abstracting the physical to logical memory mapping in the data grid, it also allows clients to access that data as if is located in a central global memory[1].

EXPERIMENT

The goal of this project is to create a P2P data sharing network using LV nodes that are capable of acquiring sensor data and publishing it on the network [6]. Currently these nodes work in a client server mode. However, because these nodes can run LVRT applications [7], we can create a P2P system where each node in the grid is aware of every other node in the P2P system, and can query the node to access the sensor data. By adding a P2P data grid, we get the following benefits:

Global Namespace

The major benefit of having a P2P data grid is that is allows internal and external nodes to access data without having any knowledge of where that data is physically located. This is done by providing a physical to logical mapping of data to a global namespace that all peers in the grid can have equal access to. External nodes can then query any peer node in the grid to locate the actual data point in the grid.

Sharing of Sensor Data Among Peers

Since all peers in the grid are aware of the data and resources available in the grid, they can use this information to minimize network traffic and share resources on need basis. For example, if one node needs more disk space it can request other nodes to share any free disk space that they are not using.

---

[1] There are many open issues with P2P and sensor networks, like network security, wireless communication, power consumption, etc. These issues are not addressed in this report.

Dynamic Network Topology and Fault Tolerance

A P2P data grid allows dynamic insertion and removal of sensors in the data grid. This way even if one sensor or node goes offline, the overall data grid can continue to function. By adding some redundancy in the system, we can also make the grid fault tolerant, such that if one sensor dies another sensor can take over its responsibilities.

Hardware

For this project, I used National Instruments' FieldPoint RT controllers ([c]FP-20xx serial product line) and LabVIEW Real-Time 7.0 software. A FieldPoint system is a modular I/O system that consists of a controller and one or more of I/O modules. The I/O modules are designed to acquire a variety of physical data, like temperature, pressure, etc. Numbers of physical channels in a FieldPoint system depend on the number and type of I/O modules used. The [c]FP-20xx controllers are also capable of running LVRT applications for embedded control and data-logging. These controllers also support a publish-subscribe protocol to publish the I/O data over the network, which allows clients to directly talk to the controller and read or write any of the I/O channels.

Although the resources and number of channels available on a FieldPoint system are slightly higher than what a typical sensor network is likely to have [4], they are ideal to simulate a P2P data grid as they have the following capabilities:

- Each node can be placed in geographically different, potentially hazardous, location
- Each node is capable of running a LabVIEW Real-Time application
- Each node has one or more sensors connected to it
- Each sensor can acquire live physical data
- Each node can share its sensor data over the network using a publish/subscribe protocol [5]

P2P DESIGN

Figure 1 shows the basic design for the P2P data grid. A peer in this grid is a FieldPoint controller with some I/O channels connected to it [6]. The nodes are capable of communicating among each other as peers and can share information among each other, such that no one node has all the information available in the grid, but when all these nodes come together as peers in the grid, they can present the I/O data as if it is coming from a central repository.

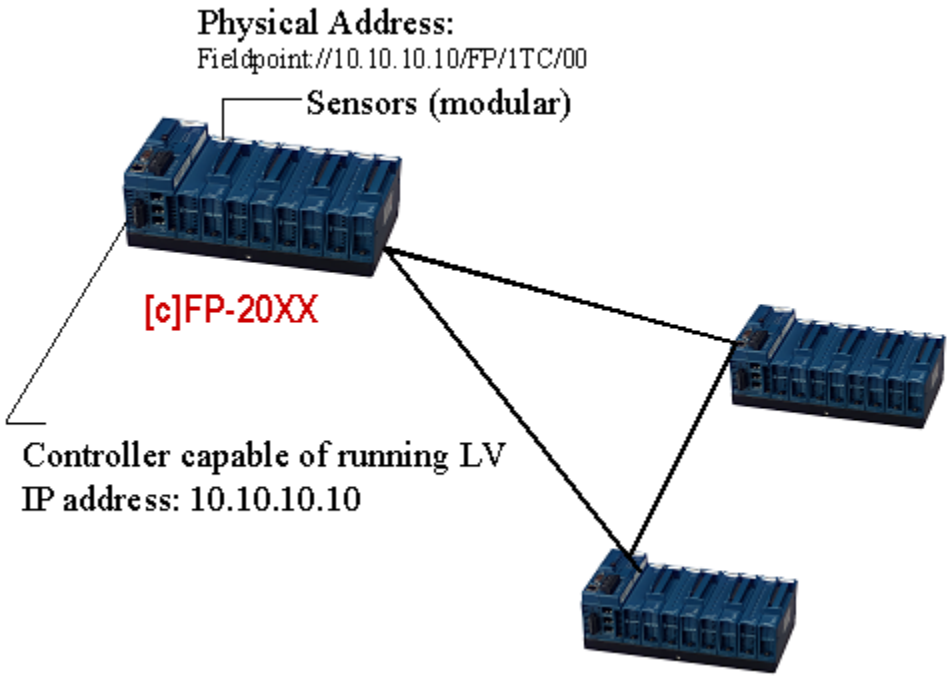

Figure 1 P2P data grid with FieldPoint LVRT nodes

The nodes share their data with each other by creating a mapping from physical address to a logical name. This logical name is unique for each channel and serves as the global name for that I/O channel in the data grid. The logical namespace is created using a default naming convention, which each node conforms to. By allowing a default naming convention, we can minimize the initialization needed by the user to setup the system. However, the system can allow the user to change the logical name to a user-defined name, once the grid is initialized. Table 1 shows an example mapping where each controller has a temperature channel connected to it. In this example, each node has specific region assigned to it (e.g. Zone1) and uniquely names all its channels based on this region and channel offset.

Physical Address	**Logical Name**
fieldpoint://10.10.10.10/FP/1TC/00[1]	Zone1/Temperature1
fieldpoint://10.10.10.11/FP/1TC/00	Zone2/Temperature1
fieldpoint://10.10.10.11/FP/1TC/01	Zone2/Temperature2

Table 1 Mapping from sensor's physical address to default logical name[2]

[2] The physical address in my setup was actually an internal logical mapping to the actual physical address that only the owner node has information about. I used the internal logical mapping because it is easier to program with the FieldPoint API [6]. However, using the actual physical address (as given in Table 1) is not too different from using this internal logical mapping.

P2P Clients

Any peer within the grid is capable of acting as a client of the shared data space. Because the node is already a peer in the system, it shares common knowledge logical namespace and can use that to find the sensor it is interested in locating. Details on how the logical namespace becomes common knowledge is discussed in a later section.

In addition to a peer node, any external computer on the network that can access the grid can become a client. However, because an external client does not have access to the common knowledge that peers within the grid share, it must know the IP address of at least one peer node in the system that can act as a communication proxy for the external client[3]. The client does not become a peer in the system to access data, but uses point-to-point communication with the proxy peer to access shared data using global logical names.

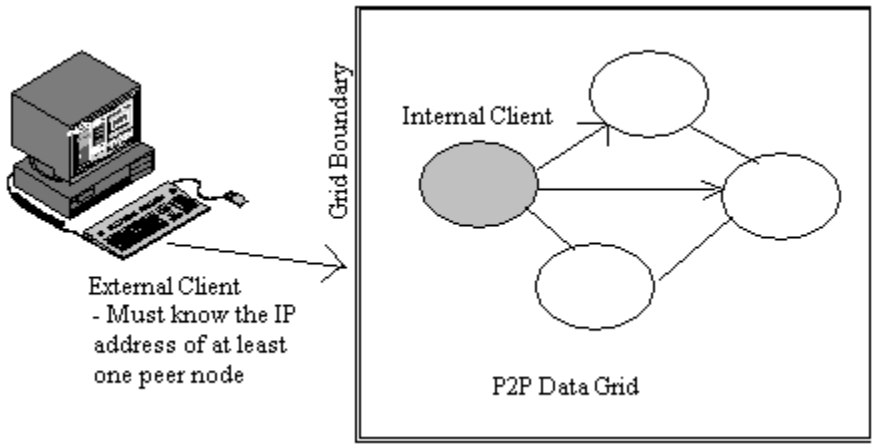

Figure 2 P2P Grid and internal/external clients

P2P Data Grid Implementation

As mentioned earlier, I used the LVRT 7.0 software to implement the data grid [7]. Most of the work is related to implementing a distributed hash table (DHT) [8] using LV's array data type and communication VIs [7]. I used the Content Addressable Network (CAN) indexing mechanism [9] to place different peers on the grid and share global namespace.

Brief Overview of CAN

In the CAN indexing scheme [9], the grid is an n-dimensional logical space and each node is randomly assigned a part of that grid space. When a new node enters, it randomly selects a point P on the grid and requests the existing nodes to let it join the grid. The nodes that already exist in the system collaborate with each other to split the grid in such a fashion that allows the new node to own part of the grid space and gives it knowledge of immediate neighbors. If the P2P system is used to share information among nodes, like file sharing, then each new file is added to the grid via a (DHT) [8] whose hash function determines the point Y where the information will be stored. A node inserting the information

[3] Alternatively, the first node in the P2P system can register itself with a DNS server using a well-known hostname that external clients can use to access the shared data.

uses this hash function to locate the same point Y and sends its information to the owner of the region where Y lies. The nodes communicate with each other by routing their messages through their neighboring nodes until the message reaches its destination. Ratnasamy, et al., give an excellent presentation of CAN in [10].

CAN in P2P LV Grid

I used CAN to create the P2P data grid, with the limitation that the grid is only 2d and could only have 255 maximum nodes. The limitation of 255 nodes was due to the fact that the nodes I used were all on the same subnet.

I used an array of **NodeElement** (Figure 3a) to store my node information. Each **NodeElement** contains the IP address of a node and the rectangular region in space that it owns [X_0Y_0, X_1Y_0, X_0Y_1, X_1Y_1]. The array of **NodeElement**'s is shared among all nodes. However, because there is an upper bound on the number of nodes in the grid (i.e. 255), the **NodeTable** is replicated among the nodes, instead of being distributed. By replicating this table, each node is capable of directly talking to the other peer node, and no routing is necessary. This scheme is different from the CAN scheme, where each node only knows its neighboring nodes, but is similar to Chord's finger table scheme [11]. Ratnasamy, et al. actually discuss a similar finger table scheme to optimize CAN routing in [12].

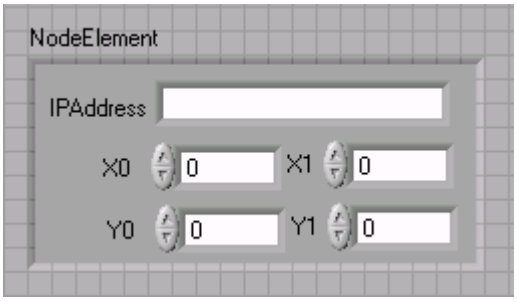

Figure 3a NodeElement data structure

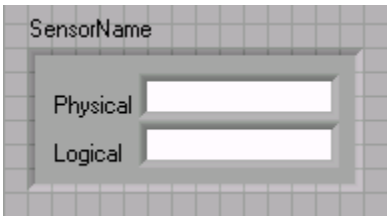

Figure3b SensoreName data structure

In addition to storing the node information, the grid also has shared a **DataTable**. This table is simply an array of **SensorName** data structure (shown in Figure 3b). A **SensorName** element contains the physical to logical mapping for a given sensor in the grid (see Table 1).

Unlike the **NodeTable** that has only 255 elements, the number of elements in the **DataTable** can be very large (255*144=36720) where 255 is the number of maximum nodes in the grid and 144 is the maximum I/O channels each node can potentially have. Because of its large size, the **DataTable** is a true DHT: i.e. information contained in this table is *distributed* among nodes, such that no one node in the grid has a complete table in its memory, but the system as a whole has the complete table that can easily be accessed by each peer node.

The **NodeTable** and the **DataTable** comprise the *common knowledge* that each peer in the grid system has access to. Details of how this common knowledge is created and shared are given in a later section.

Communication Among Nodes

The peer nodes communicate with each other via message passing. UDP is used as the underlying network protocol. Most communication among nodes is point to point. However, some messages use UDP broadcast and are sent to all peer nodes.

Table 2 gives a list of message types defined and implemented for this P2P system. Several other messages can be added to the grid system, such as Read, Write, or Heartbeat, which will mostly need point-to-point communication.

Message Type	**Uses Broadcast?**	**Purpose**
Joining	Yes	This message type is used when a new node tries to enter the grid system. Because a new node does not have any information about the existing system, it uses UDP broadcast to send its request to all existing nodes. Any node that receives this message responds to the sender node using point-to-point connection and **InitializeNodeTable** message type.
Find	No	This message is sent by the "client" to find a sensor in the grid. The client simply has to send the global sensor name in the message packet. Point to point communication is used because the **NodeTable** is replicated. Each peer can determine the point P where the sensor is located using the common hashing function and then lookup the corresponding node in the **NodeTable** to determine the owner of that grid space.
Found	No	This message is sent in response to the **Find** message. Because the sender of this message knows the node that sent the **Find** message, only point-to-point communication is needed.
InitializeDataTable	Yes	Once a new node joins the grid, it sends the **InitializeDataTable** message that contains a list of all the sensors owned by the new node. This message is sent as a broadcast message. Each node that receives this message iterates over the list and adds it to its *bucket* of hash table piece if the sensor name maps to its grid space.
InitializeNodeTable	No	This message is sent in response to the **Joining** message. As the sender knows the IP address of the node that sent the **Joining** message, point-to-point communication is used. The sender of this message the splits the grid by adding the new node and updates its **NodeTable**. It then sends a copy of its **NodeTable** the new node. Since the **Joining** message is a broadcast message, more than one one is likely to send out the **InitializeNodeTable** message to the new node. The new node only accepts the first response it receives and copies the global **NodeTable** to local memory. All other responses are ignored.

Table 2 Messages defined for the P2P LV Grid

Node Discovery and State Transition

When a new node comes up it initializes a local data table of **SensorName** elements using the sensors that are connected to the node and sets its state to *Joining*. It then broadcasts a **Joining** message with its IP address to the local subnet. Depending on the system state, there are three possibilities:

First Peer If this is the first node in the system, then it will not receive any response to the **Joining** message. The node tries three times and if it times out on all three attempts, it assumes that it is the first node in the system. With that assumption, the node changes its state to *Ready* and takes ownership of the whole grid space. At this point there is only

one entry in the **NodeTable** and no entry in the **DataTable**. If the node receives a **Find** request from an external client, it will look it up in the local data table only[4].

Second Peer If there is only one node in the system and a new node arrives (i.e. broadcasts a **Joining** message), the first node splits the grid into half, updates the **NodeTable** and sends a copy of the **NodeTable** to the new node. The new node updates its **NodeTable**, changes its state to *Ready*, and broadcast its local data table. The receiving node iterates over the data table list and adds sensors that map[5] to its grid space to its part of the **DataTable**.

Third and Subsequent Peers If there are two or more peers in the system and a new node arrives, all the existing peers split the grid and update their **NodeTable** and send a **InitializeDataTable** message with the **NodeTable** structure to the new node. The new node only accepts the very first response[6] and copies the **NodeTable** to its local copy of **NodeTable** and changes its state to *Ready*. Because all nodes use the same spit algorithm[7], the changes to **NodeTable** are the same at each peer node. The new node then broadcasts its local data table, which each peer in the system processes and adds it to its part of the **DataTable** if the entry maps to its grid space. Each node also has to update existing entries in the **DataTable** because the division of grid space has changed[8].

Finding a Sensor in the Grid

Once the grid is initialized (i.e. has one or more peers in the *Ready* state), any peer within the grid can send a **Find** message to retrieve a sensor. Since each peer has knowledge of all other peers (common **NodeTable**), it can directly go to the peer node that has the sensor in its part of the **DataTable**. Currently, the responding node only sends the physical address of the sensor to the requestor. However, it can be extended such that the responding node actually retrieves data from the physical sensor (by communicating directly to the owner) and then return the I/O data to the node that sent the **Find** message.

Fault Tolerance

I did not get a chance to implement fault tolerance, but the system can be extended to add fault tolerance by adding redundant sensors. Each element in the **DataTable** can be extended to contain not only the primary sensor, but also a list of redundant secondary sensors. Secondary sensors share the same logical name as the primary sensor, but have a different physical address. A sensor gets the primary or secondary sensor status based on its arrival order: i.e. if a sensor with name A is being added and another sensor with the same name already exists, then the new sensor goes to the secondary sensor list. If a peer node fails to retrieve the primary sensor from its owner, it puts one of the secondary

---

[4] This part is not yet implemented. In addition, the very first node currently does not add its local I/O data to the distributed **DataTable**, but that can be easily added as the first node knows it is the first node and can broadcast an **InitializeDataTable** message as soon as the second peer goes to the *Ready* state.

[5] This mapping is done using a common hash algorithm (H), which given a non-empty string S, returns point P(x, y) such that x=Hx(S) and y=Hy(S).

[6] Current implementation does not have any shared lock to protect the global information, so it assumes only one node tries to enter the system at any given time and only when it goes to the *Ready* state, another node makes an attempt to join the grid.

[7] Currently the split grid only splits the grid across the X axis, but it is easy to change that to split across either X or Y based on some condition.

[8] This part is not yet implemented.

sensors as the primary sensor and removes the original primary sensor from the list. If the original sensor becomes alive again, it needs to add itself to the grid as if it was a new node.

RESULTS and CONCLUSION

Figure 4 shows the user interface for this application. Each peer that runs this application has to initialize the **MyIPAddress** and **Region** controls. Once the node is in the *Ready* state, user can enter the globally shared sensor name and click on the **Find** button. If the sensor is found in the grid, its physical address is displayed in the **Physical Address** indicator. Appendix A has a list of all the LabVIEW VIs created for this implementation and their corresponding function.

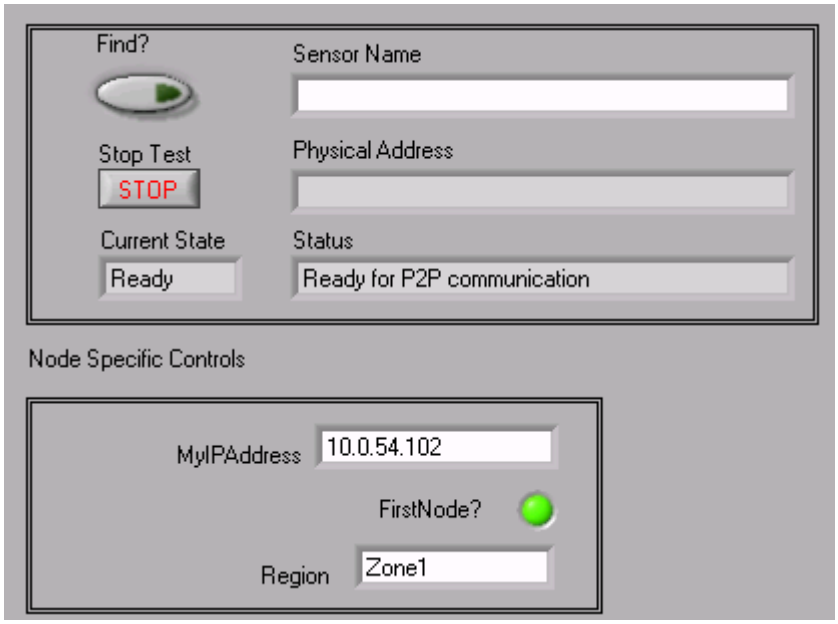

Figure 4 User interface for P2P LV data grid

Unfortunately I could not get the peer nodes to synchronize properly when they were being added to the system, so I do not have any experimental results. However, as mentioned in the implementation section, the shared global namespace is stored in CAN based DHT, so I would expect this system to be very scalable as long the number of peer nodes is limited to 255. The CAN scheme I implemented replicates the **NodeTable** and is very similar to the finger table scheme of Chord. This indexing scheme is not scalable for very large systems (that are not limited to local subnet). In that situation a combination of CAN and Chord can be used to optimize node message routing [9]. The lookup time to find a sensor based on its global name should be $O(1)$ as the peer node can determine the node that stores the sensor mapping by just looking it up in its **NodeTable**. Most of the communication among nodes is point-to-point so there should not be a lot of network traffic. Broadcasting is only used when a new node enters, which, in a real system, should not happen very frequently.

In general, I found the CAN indexing scheme to be very useful and flexible. Users are free to implement any split and hashing algorithm to make sure their system has good load balancing.

RELATED WORK

A number of references have been mentioned in this report that discuss sensor networks and P2P networks using different indexing schemes. However, I did not find any work that is specifically related to P2P data grid that shares live data using the CAN indexing scheme or P2P grid based on a DHT implemented in LabVIEW.

APPENDIX A

p2pNode.vi Main application. This VI has three threads: Thread 1 is used for initializing the local I/O data and broadcasting the **Joining** message; Thread 2 is a UDP listener threads that responds to different messages supported in the P2P grid (see Table 2); and, Thread 3 is a client thread that responds to user requests (see Figure 4).

FindP.vi This VI maps a point P to the node that covers that region.

FindSensor.vi This VI finds the node that stores the mapping of sensor's global name to an actual physical address.

SplitGrid.vi This VI splits the existing grid region that contains point P into two sections. By default, the grid is split across the X-axis.

Create/ParseMsgPacket These VIs create or parse message packets that are used by the peer nodes to communicate with each other.

convertStringto[X] These VIs convert a given string to some LabVIEW data structure X.

convert[X]toString These VIs convert some LabVIEW data structure X to a string that is used to send that data structure to other peers.

Other VIs All other VIs used are either utility VIs, FieldPoint API VI, or LabVIEW's UDP communication VIs.

REFERENCES

[1] J. Heidemann, et al., "Building Efficient Wireless Sensor Networks with Low-Level Naming", USC/Information Sciences Institute.

[2] D. Barkai, "Technologies for Sharing and Collaborating on the Net", in *Proceedings - First International Conference on Peer-to-Peer Computing*, August 2001.

[3] D. Brookshier, D. Govoni, and N. Krishnan, "JXTA: Java™ P2P Programming", SAMS, 2002.

[4] C. Intanagonwiwat, et al., "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks", USC/Information Sciences Institute.

[5] http://zdnet.com.com/2100-1103_2-5084955.html

[6] FP-20xx User Manual

[7] LabVIEW – Real-Time Module User Manual

[8] http://www.linuxjournal.com/article.php?sid=6797

[9] S. Ratnasamy, et al., "A Scalable Content-Addressable Network", University of California, Berkeley.

[10] www.icir.org/sylvia/sigcmm01.ppt

[11] http://www.pdos.lcs.mit.edu/~rtm/slides/sosp01.ppt

[12] S. Ratnasamy, et al., "Routing Algorithms for DHTs: Some Open Questions", University of California,Berkeley, http://www.cs.rice.edu/Conferences/IPTPS02/174.pdf.