CRAM: Co-operative Routing and Memory for Networks of Embedded Sensors

by

Noshirwan Kavas Petigara

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 24, 2002

Author ....
Department of Electrical Engineering and Computer Science
May 24, 2002

Certified by........
Sanjay E. Sarma
Associate Professor
Thesis Supervisor

Accepted by............ ......
Arthur C. Smith
Chairman, Department Committee on Graduate Students

CRAM: Co-operative Routing and Memory for Networks of Embedded Sensors

by

Noshirwan Kavas Petigara

Submitted to the Department of Electrical Engineering and Computer Science
on May 24, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

This thesis presents CRAM, a framework that enables the computational and memory requirements for an application to be equitably distributed among multiple embedded sensor nodes connected on a local area network. CRAM uses a *local reasoning model* to enable complex global behaviors to arise from simple component programs that are replicated across multiple nodes in the network. By allowing nodes to work co-operatively to route and process data, this framework precludes the need for a centralized control or processing entity. This prevents bottlenecks that occur in centralized systems, allows CRAM to scale incrementally as nodes are added to the network, and ensures robustness in the face of node failures. All inter-node communication in CRAM takes place through coarse-grained or fine-grained *event-based* routing. This thesis shows that by using a combination of coarse and fine-grained event routing, CRAM's processing efficiency, for any application, is at most a small constant factor less than the efficiency of a equivalent centralized application.

Thesis Supervisor: Sanjay E. Sarma
Title: Associate Professor

Acknowledgments

Credit for this thesis is shared by many people. I would like to thank:

My advisor, Sanjay Sarma for encouraging me to think in novel and non-traditional ways, and for allowing me to explore ideas other would think crazy. He has imparted much wisdom about both life and research, which I am sure will always remain with me. His untiring enthusiasm always amazes me.

Daniel Engels for his constant, but well meaning prodding, and for his extremely helpful suggestions and criticism.

Amit Goyal and Tanaz Petigara for helping me proofread this thesis. They helped me extract meaning from a mess of words.

My labmates - Amit Goyal, Junius Ho, and Arundhati Singh for for trading stories, jokes, and generally making time in lab always an enjoyable experience.

All the people at the Auto-ID center. For discussions, for their help, and much more. The little things really do add up.

Neira Hajro for her laughter and encouragement through all of my moods, both good and bad.

And last, but not least my family : my parents, Kavas and Paula Petigara, and my sister, Tanaz Petigara for always being there when I need them. Without their caring and support I would not have been able to do this.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the emergence of low-cost environmental sensors, in conjunction with identification and location technologies such as Radio Frequency Identification (RFID) tags, have enabled us to gather real-time digital representations of the physical world. At the same time, the cost of networking hardware has dropped so dramatically that network interfaces are being added to almost every device. This has presented the opportunity to connect hundreds of digital sensors in our homes and workplaces to provide a unified view of our surroundings.

While hardware advances are rapidly pushing these devices into our surroundings, current software systems are unable to utilize their potential. The client-server model, often used to control these systems, is ill-suited to large-scale deployment of these devices and applications, leading to problems with scaling, maintenance, performance, and programmability. However, the increasing computational power of these sensors has offered the chance to rethink how they operate and are programmed.

In order to create robust, powerful applications for these sensors, I envision a system that allows these devices to organize and co-ordinate their activities, and configure their behavior based on the applications they are serving. Devices that co-operate should be able to distribute responsibility for an application equitably among participating nodes and seamlessly route data among these components. A decentralized system along these lines would be able to scale incrementally as more devices are added to the network. In addition, it would be more robust and would require little human intervention for its continued operation.

1.1 Thesis Statement

In this thesis, I present CRAM - a framework that enables the computational and memory requirements for an application to be equitably distributed among multiple embedded sensor nodes connected on a local area network. CRAM allows for complex global behaviors to arise from simple component programs that are replicated across multiple nodes in the network. By allowing nodes to work co-operatively to route and process data, this framework precludes the need for a centralized control or processing entity. The decentralized nature of CRAM prevents bottlenecks that occur in centralized systems, and is key to both its ability to adapt to the addition of nodes and its resilience in the face of node failures.

1.2 Approach

CRAM is directed towards networks of a few hundred nodes, each of which controls multiple RFID or environmental sensors. Every node reads hundreds of values from its sensors during each snapshot, which can vary from once a second to once a minute. The applications directed towards these networks are complex and often include tasks such as detecting patterns across nodes, or processing the combined data of multiple sensors into a different form. While none of the nodes have the resources to assume the responsibility for an entire application, the sum total of the computing resources of the nodes in the network far exceeds the amount needed by any single application. In order to produce the desired result, CRAM automatically divides the computational and memory requirements among the participating nodes. Each node hosts a small component that performs processing tasks on the local data read by the node, in association with a small subset of the data from other nodes. By processing combinations of local and remote data, the entire system can be made to behave as a single application, giving rise to globally relevant results.

In CRAM, each application is described using a local reasoning model, which views an application from the standpoint of a single node. In this model, an application is modeled as a component that resides on a single node. This component interacts between data local to its sensors and data that is produced by some undetermined remote nodes. Thus it is not important which nodes the data comes from, rather that the data is not local to the node itself. To produce the desired global result, this component is then replicated across all nodes participating in the application. If necessary, an application can be composed of multiple heterogeneous components. This model allows components to process and exchange data in parallel, leading to an efficient utilization of computing resources.

CRAM provides the substrate to enable efficient inter-node communication in this model. To receive remote data, a component creates an event filter to subscribe to events. When a node produces these events, CRAM automatically routes the event to its subscribers based on the active filters. In CRAM, event filters can be of two types: coarse-grained and fine-grained. Coarse-grained filters are those which match large groups of data, and generally remain active for a long time. Fine-grained filters usually match only a single event and remain active for only a short period of time when the event is likely to be produced. To support both these filters types, CRAM maintains a distributed index of events; each node is responsible for storage and retrieval of a small portion of the events produced by the system. This index can be accessed from any node using a uniform addressing scheme.

The CRAM framework presents each node with a unified view of data on the network such that data can be routed and addressed without knowing its source. By providing both fine-grained and coarse-grained routing of events, this thesis argues that CRAM can efficiently distribute processing and memory requirements across a network of embedded sensors for a wide range of applications.

1.3 Contributions

- The local reasoning model, which enables complex global behaviors to arise from simple component programs that are replicated across multiple nodes in an embedded sensor network.

- The CRAM framework, which allows applications built using local reasoning to efficiently communicate and co-operate by using a combination fine and coarse-grained event-based routing.

1.4 Context

This thesis is motivated by the challenges faced in the Auto-ID Center at the Massachusetts Institute of Technology. The Auto-ID Center is researching and developing automated identification technologies and applications. The Center is creating the infrastructure, recommending standards, and identifying automated identification applications for a networked physical world. The Auto-ID Center anticipates a world in which all electronic devices are networked and every object, whether physical or electronic, is electronically tagged with information specific to that object. The center visualizes the use of physical 'smart' tags that allow remote, contactless interrogation of their contents, enabling all physical objects to act as nodes in the global Internet. The realization of this vision will yield a wide range of benefits in diverse areas including supply chain management and inventory control, product tracking and location identification, and human-computer and human-object interfaces [22].

At the heart of the Auto-ID initiative is the Electronic Product Code (EPC) [2]. The EPC consists of a 96-bit number[1] embedded in a tiny Radio-Frequency Identification (RFID) memory chip or "smart tag" [21]. Each of these smart tags is scanned by a wireless radio frequency tag reader. Once scanned, the identity code, or EPC is then transmitted onto the Internet, where it is stored in the Physical Markup Language (PML) [3], an XML based language for describing physical objects. The

[1]There is also a 64-bit version of the EPC [1].

EPC works in conjunction with a PML description and the Object Name Service (ONS) [6] to locate and store information on every item.

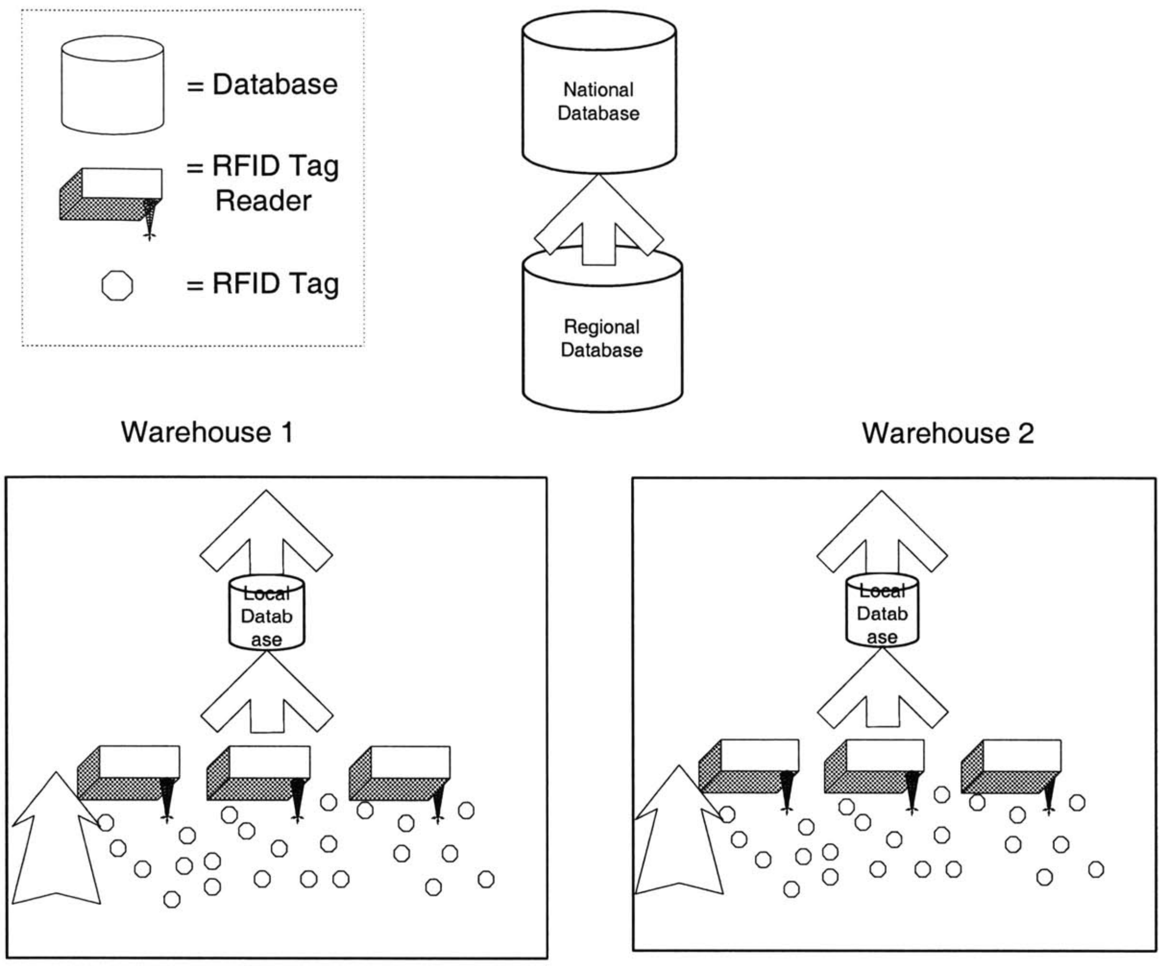


Figure 1-1: A High-Level View of the Auto-ID System

Figure 1-1 shows a typical layout of the Auto-ID system. Multiple readers are positioned throughout warehouses and supermarkets. The readers record the current state of all tagged objects in real time. At this level, there might be multiple applications running, e.g filtering, theft detection, and inventory control. Filtered information (perhaps containing only daily totals) is sent upwards to the next highest level where it is aggregated with data coming from different warehouses. This process continues until the data reaches the highest level in the system. Each level contains information on the state of the data at a different time scale. In the warehouse, data is recorded on a second by second basis. At the regional level, only daily changes are recorded, while the highest level might contain weekly or monthly totals. The lowest level of the system, or the edge of the network, is the most critical piece in this system. It is at this layer that most of the processing occurs. First, raw and noisy data is filtered. This data is then passed through multiple applications to perform functions such as theft detection and re-shelving alerts.

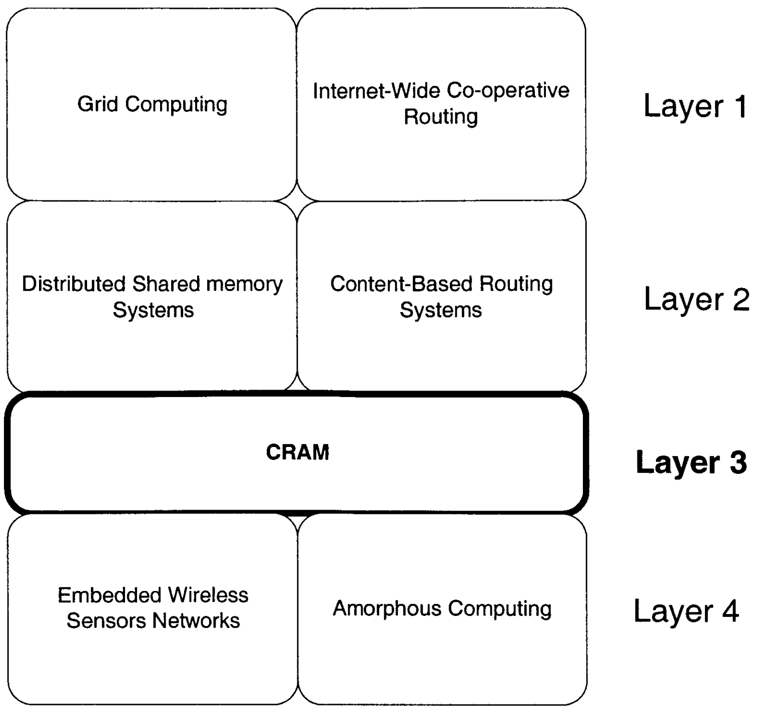
One mechanism to perform processing at the edge of the network is to centralize all the functionality onto a single 'edge' server. Each reader would send its data to this server, which would then perform all necessary processing. However, this approach is not feasible as it is unlikely that a single edge server would be powerful enough to handle the large volume of events produced at this level. Additionally, this approach does not scale well when new readers or applications are added to the system, and is also a single point of failure for the system.

In order to solve this problem, this thesis proposes an approach where computational resources for each applications are divided among participating readers (nodes) in the network; nodes co-operatively route and process data without the need for any centralized control. This approach prevents the bottleneck presented by a centralized approach, scales incrementally as new nodes are added, and is more robust in the face of failures.

1.5 Related Work

This thesis lies at the intersection of many bodies of work. The systems related to CRAM are grouped into 4 layers, as depicted in figure 1-2, according to the scale at which they operate. This hierarchy is bounded at the top by layer 1 systems, which work on an Internet-wide scale, and at the bottom by layer 4 systems, which operate on a micro-scale. CRAM occupies at level 3, a niche that is below layers 1 and 2, but above layer 4.

- Layer 1: Comprises of Internet-wide systems for distributed computing and co-operative routing.



Grid Computing
Internet-Wide Co-operative Routing
Layer 1
Distributed Shared memory Systems
Content-Based Routing Systems
Layer 2
CRAM
Layer 3
Embedded Wireless Sensors Networks
Amorphous Computing
Layer 4

Figure 1-2: Related Work Grouped into Layers. Higher Layers Indicate Larger Scale of Operation.

The Grid Computing initiative aims to harness the computing resources of workstations and servers distributed across the Internet in order to provide a scalable, robust, and powerful computing system that can be easily accessed [7, 8]. In this respect, CRAM is very similar to these systems except for the scale at which it operates. However, there is one crucial difference - most distributed computing systems aim to take data that resides at one place and distribute it efficiently across the network for processing. Nodes in a sensor network, however, *are* the sources of data. Consequently, CRAM leverages the fact that the data is already distributed to provide a mechanism such that data can be processed *in situ.*

Chord [29], Tapestry [14] and Pastry [19] are systems that achieve decentralized co-ordinated location and routing. CRAM shares many similarities with these systems, especially to Chord in its use of Consistent Hashing [13]. However, these systems are optimized for Internet-wide operation and consequently opt for scalability at the expense of performance. Therefore, these systems are unsuitable for real-time processing of data, In addition, these systems are usually targeted towards peer-to-peer file systems and persistent file stores [4, 14].

- Layer 2: This layer includes two distinct system types - the first type targets clusters of workstations or servers to provide a shared view of memory, while the second type includes content-based routing systems. Both these types are placed at this level as they are generally targeted towards clusters of workstations or servers connected on a local area network.

  For its distributed index, CRAM draws from distributed shared memory (DSM) systems such as Linda [9, 10], to provide the ability for applications to access data that reside on different nodes. However, the role of distributed shared memory is very different from these systems. Most of these systems use DSM as a programming abstraction such that the programmer cannot see the difference between local and remote memory access. However, CRAM differentiates between these two types of memory - the aim of the distributed query capability is not to mask these differences, but to provide a single addressing scheme for all remote memory.

  The other area in this layer is filled by content-based routing systems. While content or event based routing is a major component of CRAM, its design goals deviate significantly from previous work. Content-based routing systems such as Elvin4 [23] use a single server, or group of servers, to perform publish and subscribe operations. This runs contrary to the vision of a totally distributed routing system. While previous work has been accomplished in the area of totally distributed content-based routing, such as Scribe [20] and Bayeaux [32], these are built upon systems from layer 1 and hence share the same differences with CRAM.

- Layer 4: The lowest layer of related work comes from two areas. The first area is wireless sensor networks [18, 5]. Systems such as SCADDS [5] have the same goals as CRAM. However, in these systems, a node cannot communicate with all nodes in the network, but only with a handful of nodes within radio range. These systems use ideas such as directed diffusion [11], content-based routing schemes developed to overcome these limitations. The second area at this layer is characterized by the Amorphous Computing paradigm, which has provided the inspiration that complex global behaviors can be developed from local interactions of adjacent nodes. However, CRAM has a very different notion of 'locality'. As CRAM operates on a wired network, any two pairs of nodes can communicate. Hence each node can is assumed to be 'adjacent' to every other node. CRAM uses this augmented definition of locality in its local reasoning model.

CRAM is designed to operate in an environment that consists of multiple nodes, each of which is delegated the responsibility for multiple sensors. Such systems are at the core of the Auto-ID Center architecture; a single reader is responsible for reading hundreds of RFID tags, and is also in control of multiple environmental sensors. However each reader must be able to co-ordinate its activities with other readers present in a warehouse.

1.6 Outline

The next chapter provides the design rationale for CRAM. This chapter, develops the local reasoning model, and argues that a combination of coarse and fine grained event routing leads to efficient inter-node data sharing and co-ordination. Chapter 3 details the design and implementation of CRAM. Chapter 4 provides examples of how applications can be built on top of CRAM, and chapter 5 presents conclusions and directions for future work.

Chapter 2

Design Rationale for CRAM

This chapter describes the design rationale for CRAM. The design of CRAM is primarily influenced by the observation that centralized methods for data processing are not viable for embedded sensor networks. Consequently, a case is made for decentralized, distributed processing, in which nodes co-operate to process and route data. Based on the properties of embedded sensor networks, the *local reasoning* model is described. This is used to distribute processing tasks in these networks. In order to facilitate inter-application communication, this chapter advocates the use of *transformational interactions.* The underlying routing substrate for these models is provided by a data-driven mechanism, *event-based communication.* Lastly, this chapter argues that a combination of fine-grained and coarse-grained event routing provides efficient distribution of resources across all nodes in the network.

2.1 The Case for Decentralized Processing

The first embedded sensors had limited processing capabilities and highly restricted connectivity options. They read data from the environment at a slow rate and passed this data to a host computer via a serial or parallel interface. More recently, these devices have been equipped with network interfaces. In this configuration, devices read data from the environment and send it along the network to a central server for processing. This model does work in limited settings. However, it is unsuitable for wide-scale deployments for the following reasons:

1. **Absolute processing power:** There are numerous visions of a networked physical world in which a single room might contain hundreds, or even thousands of sensors [5, 22]. The Auto-ID Center, believes that a "standard" deployment in a warehouse or supermarket will incorporate a few hundred RFID tag readers, each of which read a few hundred tags per second. In these deployments, close to a hundred thousand items of data are produced each second. At a minimum, a single server would have to read this data, filter it to remove environmental noise, and perform tasks such as theft detection and re-stocking alerts, among many others. In many cases, commodity hardware will not suffice to handle the large number of transaction and applications. This has prompted a search for alternative approaches that will yield a higher absolute processing power, at lower cost.

2. **No incremental scaling:** Approaches that involve a centralized server approach do not scale well. The addition of more nodes to the network increases the load on the server, without a corresponding increase in processing capability. Consequently, upgrading or replacing the server is the only solution when the load increases beyond the processing capabilities of the server.

3. **Single Point of failure:** A single server represents a single point of failure. Since all processing tasks are performed on the server, the failure of the server will render the entire network useless. Alternatively, a backup server is needed, which is economically unviable.

For these reasons, this thesis postulates that a centralized, single server approach is not a viable solution for the control and processing of data from embedded sensor networks. Instead, processing of data should be performed in a decentralized manner, with each node performing part of the processing. Due to the large number of networked nodes, the sum total of the processing power of all the nodes in the network is sufficient to handle a large number of applications. Additionally, adding new nodes to the networks will incrementally scale its processing power since the new node will participate in processing data. Finally, a decentralized system will be able to tolerate failure at multiple points, as each point only processes a small portion of the data.

2.2 Distributed Processing

In the previous section, it was argued that a decentralized, distributed approach to data processing is a better alternative to centralized processing in sensor networks. However, current techniques for distributed computing are not tailored to embedded sensor networks. This section first discusses general methods for distributed and parallel processing. Next, based on observations, it is argued that these methods are often not applicable for sensor networks. Based on these observations a simple model for distributed processing in these networks is formulated.

2.2.1 Distributed and Parallel Processing in Conventional Networks

Distributed computing projects such as SETI@Home [24] have gained widespread attention in recent years. The goal of these projects is to harness idle CPU cycles and storage space of tens, hundreds, or thousands of systems networked across the Internet to work together on a particularly processing-intensive problem. These projects draw from parallel processing techniques, which employ multiple processors at the same location, and have been used to build high performance computers for decades [15].

Both distributed and parallel computing have the same goal: to build high performance computing systems by processing data in parallel, on multiple processors. In general, the method to achieve this is to break the data into smaller components, each of which is processed simultaneously on a separate processor. All processors might execute the same task on their assigned dataset (Single Instruction, Multiple Data - SIMD [15]), or each processor might execute a different task on its dataset (Multiple Instruction, Multiple Data - MIMD [15]). Additionally, there are different ways to divide the data into groups for each processor. Data may be broken up such that there are are no mutual dependencies, in which case separate processors do not have to communicate among each other. However, if the datasets are broken up such that there are mutual dependencies, then the processors must be able to communicate results to each other.

While there are many factors that affect the performance of distributed processing applications, two that have interesting implications for sensor networks are discussed below.

- **Compute-to-data Ratio:** When dividing data into components to distribute across a network for computation, the first parameter examined is usually the compute-to-data ratio. Data for distributed applications usually resides in a single place, such as a database, and must be transferred to the different nodes for processing. After the computation, the results are transferred back to the source. If the latency and bandwidth of the network are high, relative to the computing cycles needed to perform the operation, then the cost of the transfer will dominate and the processing will be inefficient. Consequently, typical applications for distributed processing involve datasets that require a lot of processing per unit of data. This prevents large amounts of data from being sent to a single node at any one time, and keeps the transfer overhead relatively low.

- **Compute-to-communication ratio:** The second parameter that must be examined is the compute-to-communication ratio. When distributing tasks across the network, one needs to load balance these such that all processors are processing tasks concurrently. This prevents processors from sitting idle, and thus allows maximum utilization of computing cycles. To do this, an attempt is made to divide a task into sequential blocks that can be executed independently of each other. However, for many applications, it is very difficult to divide the tasks and data into these blocks. In general, the size of the task increases as the blocks are made more independent of each other. This creates a problem as a task might be too large for a single processor. To counter this effect, tasks and data are broken down such that they are coupled with other tasks. This creates the need for processors to communicate with each other to exchange results during a computation. As inter-node communication speed is generally slow, distributed applications always try to minimize the compute-to-communication ratio in order to keep this overhead low.

2.2.2 Distributed and Parallel Processing in Sensor Networks

While many of the same principles hold for distributed processing in sensor networks, there are significant differences that are leveraged in this thesis.

Observation 1: The data is already distributed at the nodes: As the nodes control sensors, the data is already readily available at the node itself. In traditional distributed systems, there a cost is associated with transferring the data to and from the processing nodes. However, with embedded networks, no cost is associated with transferring the data to the node for processing as nodes *are* the source of data. Therefore any processing performed at the source, no matter how small, will be effective in utilizing computational resources

Observation 2: Spatial and temporal locality of data at nodes: The other observation of critical importance is that the data that resides at each node is divided into loosely coupled partitions. As described in section 2.2.1 a significant amount of work in distributed computing systems goes into dividing tasks and data such that they are small, nearly sequential blocks that execute on each node. If data in these sensor networks were randomly distributed, then there would be far too many mutual dependencies between the groups, and the communication overhead would be tremendously high. Fortunately, the distribution is not random; it is divided into partitions that are spatially and temporally grouped. The reason that these groupings occur is simple. A single node controls multiple sensors, each of which are close to the node. As a result, these sensors are spatially related. Similarly, each sensor is read serially, which makes the data temporally related. This allows the inherent parallelism of these datasets to be exploited in order to perform operations concurrently.

These observations have prompted the development of a simple model of distributed processing, tailored to embedded sensor networks, which is described in the next section.

2.3 A Model for Distributed Processing in Sensor Networks

2.3.1 Modeling a Distributed Application: Local Reasoning

Our model for distributed processing is based on the observations outlined in the previous section. The key to this model is to conceptualize an application in terms of *interactions* between local and remote data. Local data is that which the node has gathered from its sensors. Remote data are those items available from some other node. In this model, the task of the programmer is to think of applications in this way. The burden of actually sharing the data needed for the interaction is placed on the routing layer.

For example, a typical application for sensor networks is to detect the same condition among a group of neighboring nodes (e.g. a sharp rise in temperature in an area, signaling a failure in the cooling system). In this case, one would think of the application as calculating the gradient from the local temperature sensor [the local data], and then comparing it to gradients from surrounding nodes [the interaction with remote data]. This application can then be replicated across all nodes in the network. As is intuitive, the detection of a cooling system failure takes place in parallel across the network - each component performs the same computation, but on different sets of local and remote data. Each node computes its local temperature gradient. The routing layer handles the transfer of this gradient between the neighboring nodes, which compare the remote to their own locally processed data. If any node detects an anomalous gradient difference, it can signal a problem. Although, in the case of this application, there is a single component that is replicated across all nodes, it is possible for an application to have multiple heterogeneous components, each of which is executed on the relevant nodes.

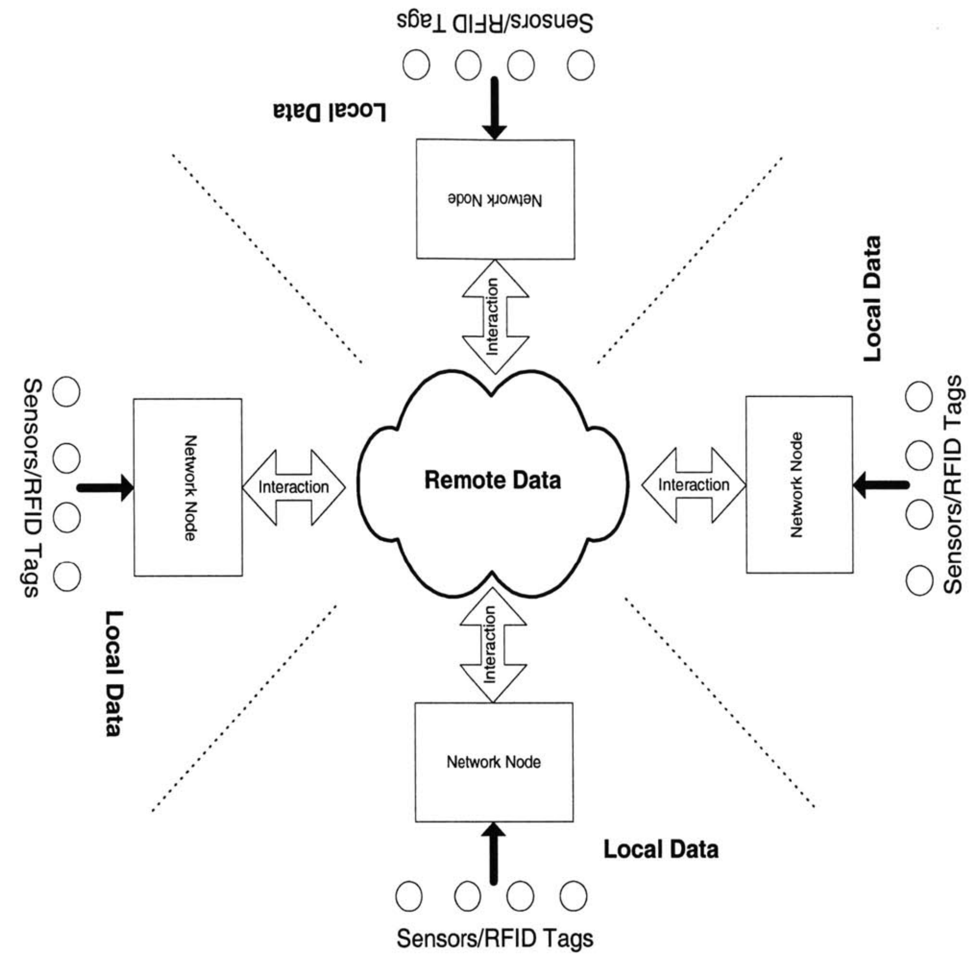


Figure 2-1: A Distributed Processing Model For Sensor Networks

While this example is very simple, it illustrates the core philosophy of the model:

- The data is already grouped and distributed at the processing nodes.
- Applications are thought of as interactions between local and remote data.
- These interactions are replicated on many nodes to allow for concurrent processing of an application.
- The underlying routing layer handles data sharing between nodes.

This model is based on the assumption that each node has enough computing power and memory to process data that is local. If each node did not have even this capability, then it would be very difficult to apply this model, as another node might have to share the responsibility for processing 'local' data. However, for the networks in the Auto-ID system the nodes have sufficient processing power to process data read from their sensors.

2.3.2 Communicating Between Distributed Applications: Transformational Interactions (TI)

Local reasoning, presented in section 2.3, provides a model for inter-node co-operation in order to distribute a single application across multiple nodes. However, this model does not incorporate a notion of how multiple applications on the same network will co-operate. Inter-application communication is crucial to these networks as they are expected to perform multiple tasks.

In order to facilitate sharing of data between co-operating applications in embedded sensor networks, a new model of data flow is presented: *Transformational Interactions.* This model is driven by the expectation that the data needed by high level applications in the network is a processed version of data needed by lower level applications. TI is a bottom-up model of data flow, in which data is viewed as being processed incrementally as it travels (logically or physically) from the outer fringes of the network into its upper layers. Each application (or component) receives local or remote data in a certain state. After processing this data, the component produces a certain output, which is viewed as a transformation of the original state. Most often, higher level applications do not need data in its raw form, but need data that has undergone processing at a lower level. Transformational interactions incorporates this characteristic of sensor networks directly into its model. TI fosters a system, where smaller applications combine to produce bigger applications. Moreover, it promotes efficient use of resources as intermediate results are available, even if they are produced by different applications. Interestingly, TI inherently integrates both the routing and processing of data packets: as packets are routed through the system, they are incrementally processed.

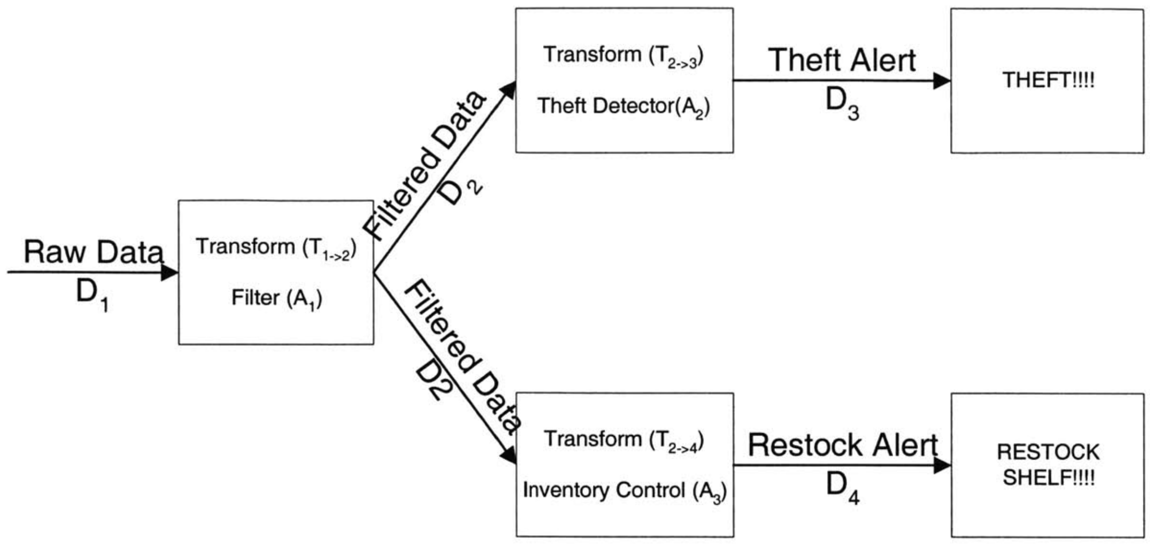


Figure 2-2: Transformational Interactions

To understand how transformational interactions would work in practice, an example of a network of RFID tag readers in a supermarket is illustrated in figure 2-2. Each reader (A_1) produces some raw data (D_1) consisting of the RFID tags that they sense within their antenna fields. A filtering component resides on each reader, which performs transformation $T_{1->2}$ on D_1 to produce data D_2 that is free of environmental noise. In addition, there are two other applications active on this network - inventory control (A_2) and theft detection (A_3), which are distributed among the nodes. Both of these applications cannot operate on raw data, but need filtered data. Rather than have each application receive and filter data raw data independently, both applications build off of the filtering application. As their input, they receive the filtered data D_2. A_2 and A_3 might apply further transformations to D_2 to produce, for example, re-stocking alerts D_4 and theft alerts D_3. Again, other applications can use these alerts to perform higher level tasks without needing to perform all the intermediate processing.

The consequence of a transformation-based approach is that the next destination for a set of data elements is based on the result of the transformation to it. As is intuitive, this method combines both routing and processing. Modeling data flow based on transformational interactions shifts the focus of programming an application - the programmer does not need to think of how and from where he is going to receive the data, but only needs to understand what state he needs the data in. This is a powerful abstraction that facilitates efficient inter-application sharing.

2.4 Routing in a Distributed Sensor Network: Data-driven Communication

The previous sections have developed a model for distributed data processing in sensor networks. However, the underlying communication mechanisms needed to support these models have not been defined. This section argues that data-driven communication is well suited to distributed processing in sensor networks. Specifically, the use of event based routing for communication is advocated.

In a client-server environment, although the number of clients might be very large, each client needs only to communicate with a small set of servers. Generally, even if information is shared between clients, the server acts as a middleman between the clients. As data is centralized at the servers, every client knows where it can retrieve all the data necessary for processing. However, in a decentralized system, the data needed for processing can reside on any node. As the number of nodes is often extremely large, it is not feasible to know where each piece of data resides. In order to overcome these problems, the local reasoning model and transformational interactions implicitly include a data-driven approach to communication. Data-driven communication is not based on the source or destination addresses of the nodes, but on the actual content of the data. This allows nodes to access data that resides on other nodes by simply specifying the content of data they need to receive.

While this argument is valid for all systems in which data is distributed across many nodes, a data-driven approach is even more appropriate for sensor networks for the following reasons:

- **"Liveness" of data:** The data from sensors in a network is dynamic, and has different meanings for different applications in the network. For example, in a warehouse, data from an exit reader is needed in real-time for theft detection. However, for a higher level application, this data is needed at a lower resolution only as a record for inventory control. This is very different from data such as file transfers and web pages in a client-server environment. In these environments, the data has only one meaning and is needed at only one point (the client or the server). Data driven communication can capture these liveness properties, by only routing specific instances of data as needed.

- **Importance of aggregation:** In a distributed sensor network, relevant data is often spread across multiple nodes. More importantly, data from individual sources is often not important unless aggregated from spatially or temporally related sources. For example, to filter out environmental noise from reader data, we need to process historical data from a single reader, as well as data from nearby readers. Data-driven communication can encapsulate these aggregations as data is routed according to its contents, rather than its source node.

2.4.1 A Mechanism for Data-driven Communication: Event-based routing

Data-driven communication is often referred to as content-based routing. Content-based routing defines a broad approach to data-driven communication, where data packets are routed based on the information they contain. In his thesis, Sheldon defines content-based routing and uses it for Internet-wide information discovery and retrieval [25]. More recently, an instantiation of content-based routing, known as publish-subscribe has been used in a number of academic and commercial systems. Systems such as Elvin4 [23], Gryphon [17] and XMLBlaster [30] use publish-subscribe to allow users or application to specify which portions of a data stream they would like to receive. Snoeren et al used a publish-subscribe mechanism in overlay networks to provide intelligent content pruning for reliable data distribution [27].

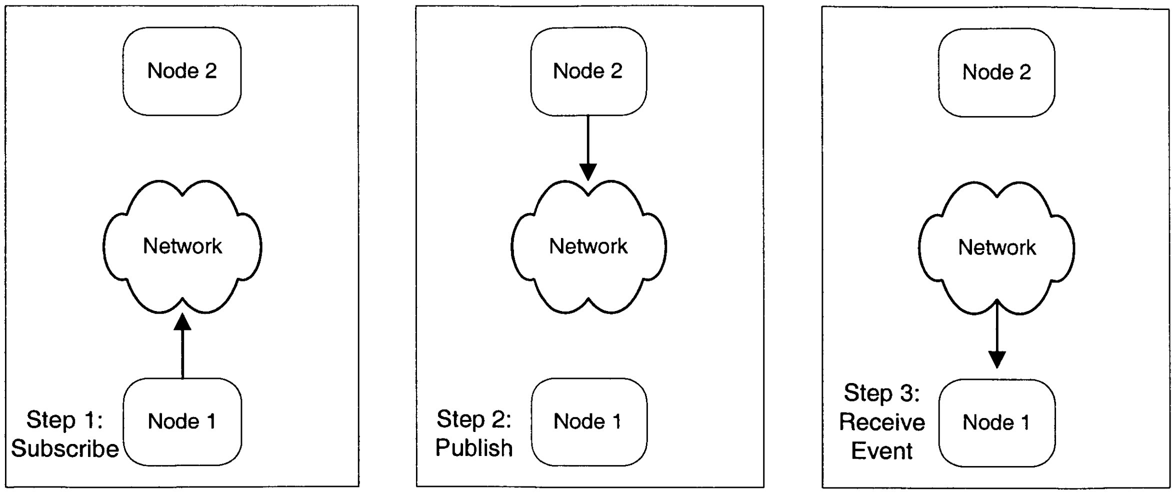


Figure 2-3: Mechanisms in Publish-Subscribe

Broadly, publish-subscribe systems work as follows:

- A client or application submits a query to routers [subscribe]
- Other clients or servers send data packets along to the nearest router [publish]
- The router then applies the query to all incoming packets to determine which packets to send to the subscriber.

Publish-subscribe systems are interesting in the context of sensor networks as they are a good match for the data-driven communication in sensor networks. In the systems listed above, publish-subscribe is used mainly as a mechanism to prune data streams. However, we envision publish-subscribe as a mechanism to identify and route *events* based on their current state. Consequently, the term *event-based routing* is used from here on to refer to these systems. In a sensor network, every atomic data element that is produced in the network is referred to as an event. For example, a reading from a sensor is an event. Additionally, processed data produced by an application also consists of events[1].

An event-based routing system would integrate with the local reasoning model as follows:

[1]The grammar and semantics of these events are described in section 3.1

1. An application would be defined as a series of components that implement local-remote interactions as described in the local reasoning model.

2. For each interaction, the component sends out a subscription request for the desired events from remote nodes. As an example, a subscription might contain a request for temperature events greater than 40° C. In this case, the component would only receive events that satisfied that criteria.

3. Each component receives events and processes them in conjunction with local events, which produces new events.

4. If these new events match any current subscriptions, they are sent to the subscribing components.

2.5 Putting it all together: Developing an Efficient Event-Routing System

In the previous sections, we described a model for distributed processing in embedded networks. This model was based on a data-driven communication model based on event-routing. Still, there are a many open questions still remain on how these models can be translated into a practical design.

The characteristic that data is already grouped and distributed to the nodes has important implications on how data is shared between nodes. As the data is already partitioned into groups, each application will view these groups as being coupled differently. Therefore, some applications will require very little inter-node sharing, while some will require a much higher level of sharing. Using conventional methods of distributed computing, data is partitioned differently for each application. However, the local reasoning model requires that each node only processes data local to itself, and thus data cannot be reorganized. In order to allow local reasoning to work efficiently across multiple application types, the underlying event-routing mechanism must be able to support all types of applications. To do this, this section first parameterizes applications based on the sharing characteristics of their components. Next, it is argued that to allow efficient division of computational resources, two distinct types of event subscriptions are necessary: coarse-grained and fine-grained. Lastly, this section shows how efficient distribution of resources is possible if both methods are provided to applications.

2.5.1 Characterizing Applications

Section 2.3, outlined a model for distributed processing in sensor networks. In this model, network wide operations are thought of as interactions between data local to the node, and data that reside elsewhere on the network. As data is in fixed groupings, the number of these interactions necessary varies greatly depending on the application. In order to provide underlying routing mechanisms that can support wide ranges of data sharing, it is necessary to first parameterize, classify and understand these interactions. In this section, parameters are defined to characterize these interaction. Applications are classified based on the proportion of nodes that each node has to co-operate with.

To examine the parameters, a simple model of the network is utilized. The network consists of grid of m nodes. Each node controls one sensor, which produces events at the rate of one event per second. Although simple, this model captures the important properties of these networks.

There are two variables that define how applications share data.

- Scope (n) – This variable describes the total number of nodes that are involved in an application. For example, in an application where the purpose of the application was to detect footsteps anywhere in a room, then the scope would be equal to the total number of nodes in the room.

- Degree of coupling (c) – This variable describes how many remote components each component needs to interact with. For example, in an application where a footstep is detected at a node if all 6 neighboring nodes detect a change in pressure, then the degree of coupling, $c = 6$. However, if the purpose of an application was to detect duplicate RFID tags from any two readers in the room, then each component would be coupled with all the components in the room and $c = n$ for that application.

Based on the these variables, the sharing ratio, d for each application, $d = c/n$, can be calculated.

- **Example of an application with small d:** Take an example of a room with 100 nodes (sensors) uniformly distributed across the floor. In order to detect a footstep anywhere in the room, we saw that $n = 100$, $c = 6$. For this application, $d = 6/100$. This application is distributed by having code on each sensor that receives data from its 6 neighbors and checks to see if all have registered a change. Consequently, if there is a footstep anywhere on the floor, it will be detected. In applications with a d that is small, each component will only have to share data with a small proportion of the components in the network.

- **Example of an application with large d:** Take a similar arrangement of a room with 100 sensors. In this case, the application needs to detect duplicate RFID tags from any two nodes in the room. Consequently, $n = 100$ and $c = 100$. In this case, $d = 100/100$. In applications with d close to 1, each component needs to share data with a large proportion of components in the network.

2.5.2 Event-based Routing: Coarse-Grained

A traditional publish-subscribe system can be classified as coarse-grained. Components subscribe to a class of data, wait for the data to be routed to them, and then process all of it. Coarse-grained subscriptions usually remain active for a long time, and are most often sent at the start of execution. This method of data driven communication works efficiently for applications that have a low sharing ratio. As this ratio rises i.e. when a component needs to share data with more components, coarse-grain event routing becomes more inefficient.

To understand this effect, we examine a simple application in this section. The network is comprised of a grid of m nodes, and the scope of the application is all the nodes in this network. Each node controls one sensor, which produces one event per second. Each event is a random number and the purpose of the application is to detect duplicate events. The application is modeled using the local reasoning model: each node hosts a component that compares its local event to remote events to detect duplicates. We vary the degree of coupling to examine its effect on the computational efficiency of the network. The computational efficiency is calculated by comparing the total number of comparisons performed by all the nodes to the total number of comparisons that would be performed in an equivalent centralized application. As the centralized application will give the best possible performance, this shows how efficiently CRAM is able to divide computation across the network.

If this application was performed in a centralized manner, each node would route its events to a central server. In the centralized case, it is intuitive that the number of comparisons performed is m, as that is the total number of events generated per second (one comparison for each event received [2]).

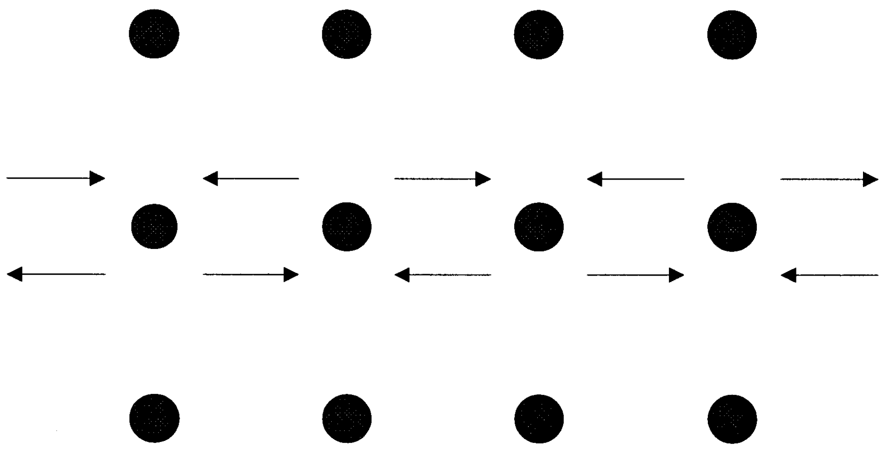


Figure 2-4: Input-Output Correspondence in Coarse-Grained Routing

For the decentralized system, when the degree of coupling is 1, each component receives events from only one other component. In an an application with homogeneous components, as each component receives events from one other component, it

[2]This can be implemented using a hash table

will be required to its send events to one remote component. It then follows intuitively , that the computation is 100% efficient. For each event, only one comparison is made because each component sends its events to only one component. Consequently, the total number of comparisons performed in the network is m. The decentralized system is as efficient as the centralized system in this case.

However, when the degree of coupling increases, the efficiency of the decentralized system decreases. In the case when the degree of coupling is 2, each component will receive events from two other components. As explained in the previous example, with homogeneous components, as each component receives events from two components, it will also send events to two different components. Thus each event is propagated twice into the network, and so for each event, two comparisons are performed.

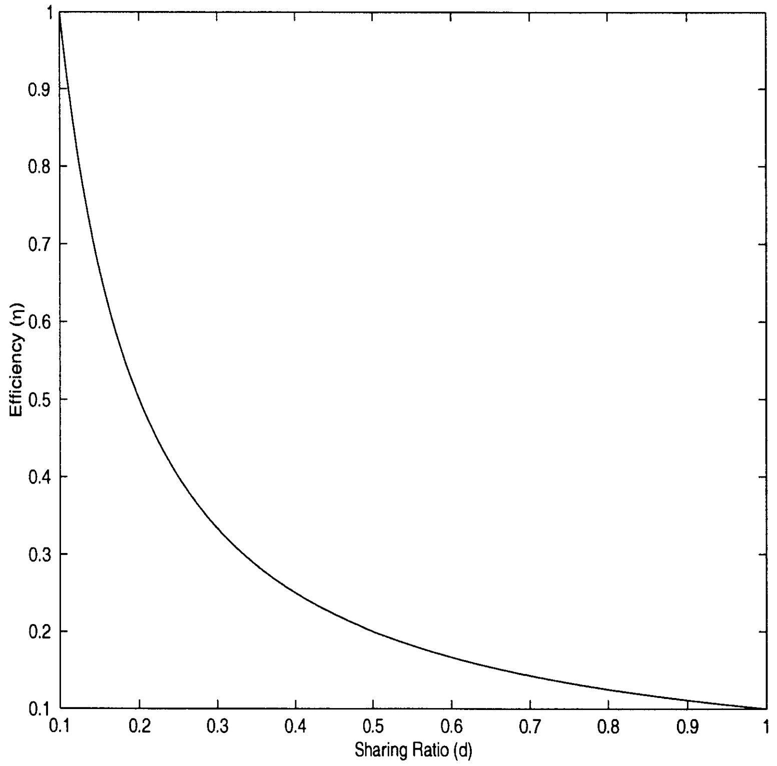


Figure 2-5: Efficiency of Coarse-grained Routing

The correspondence between the number of inputs and outputs from each component is depicted in figure 2-4. Due to this correspondence, the efficiency of coarse-grain routing mechanisms decrease linearly as the coupling of the components rises. Consequently, the generalized efficiency of a coarse-grained routing can be given by the equation $\eta = O(\frac{n}{c}) = O(\frac{1}{d})$. This relationship can be seen in figure 2-5. As is obvious, this method is not efficient for applications that have large degrees of coupling. If an application had coupling between all of its components, then *each* component would be performing the same amount of processing as a single centralized version of the application.

2.5.3 A Fine-grained Approach to Event-Based Routing

The inefficiencies of coarse-grained event-based routing for certain applications prompted a look into how sharing is achieved across multiple nodes. The ideology behind coarse-grained routing is to have all the necessary events routed to you, and then perform all the computation on the node itself. However, when degree of coupling is large, every component needs to process every other component's events. Thus each event is processed many times in the network. It would seem then that for applications with a large degree of coupling, a more viable approach would be for the component to send its events out into the network, and have the 'network' do the computation only once for each event.

In event based routing, the computation that has to be performed is a comparison - events need to be matched to filters. Consequently, a viable implementation of this 'network' computation is a distributed index in which each node has the responsibility for matching e/n events, where e is the total number of events generated in the system. In this method, nodes would dynamically send subscription queries into the network as needed. If there is a matching data item when the subscription is sent out, it would be dispatched to the subscriber, otherwise the query would be discarded. Event routing of this type is named fine-grained routing due to the types of event subscriptions are dispatched. This dynamic nature of querying for events allows components to dynamically generate subscriptions based on the local data that they receive. For example, in the application to detect duplicate events explained in the previous section, a component would read local events, and dispatch fine-grained subscriptions that match each of these events to the network. If there is a matching event (i.e a duplicate event), the network would respond with that event. Consequently, the work of matching events is distributed among all the nodes in the network.

For an application whose components have a large degree of coupling with all other components in the network, the total number of comparisons performed is m as one comparison is made per event. Thus for applications with high coupling, fine-grained event routing is as efficient as the centralized case.

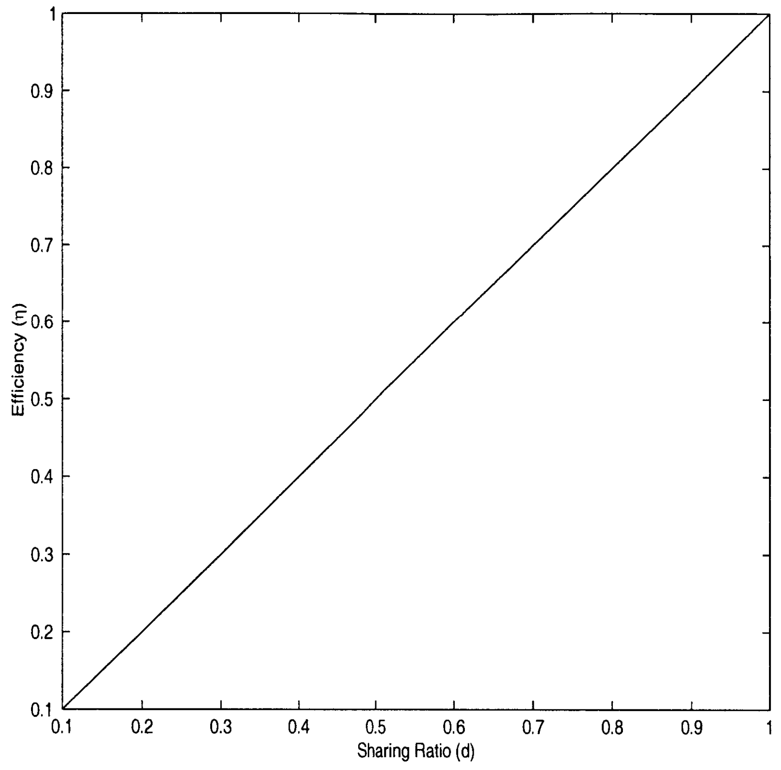


Figure 2-6: Efficiency of Fine-grained Routing

However, for applications with low coupling, fine-grained routing is found to be very inefficient. The total number of groups of coupled components can be given by $\frac{n}{c}$. When a fine-grained query is sent out, that query is matched to all other events in the network, even if the coupling is very small. The number of matching comparisons is m. However, if a component sends out a fine-grained query, it will get responses that pertain components that it is not coupled with. This is because with fine-grained routing, the component loses the ability to selectively interact with a set number of other components. More comparisons need to be done to discard the irrelevant replies. Thus for any reply to a fine-grained query, a component will have to perform $(\frac{n}{c} - 1)$ extra comparisons to discard irrelevant replies as there are $\frac{n}{c}$ groups that these replies can be sorted into.

Consequently, as the coupling increases, the efficiency of fine-grained routing also increase. This increase in efficiency can be given by the equation $\eta = O(\frac{1}{n/c}) = O(d)$. The relationship is show in Figure 2-6.

2.5.4 A Combined Approach

This chapter formulated an argument for a decentralized, distributed approach for computing in wireless sensor networks. Based on observations, a model was developed for distributed computing in these networks that leveraged their unique characteristics. Lastly, it was argued that the sharing of data between nodes could be accomplished using coarse and fine grained event routing.

As shown in section 2.5.2, coarse-grained event routing is efficient when the sharing ratio is low, while fine-grained event routing is efficient when the sharing ratio is high. In order to allow applications with widely differing sharing ratios to co-exist efficiently, the routing layer is allowed to perform either of these methods, depending on the application. Applications with low degree of coupling will use coarse-grained event routing, while applications with high degree of coupling will use fine-grained event routing.

Figure 2-7 shows the result of this approach - it is ensured that for any application the efficiency of processing is at most a small constant factor worse than the efficiency of an equivalent centralized application.

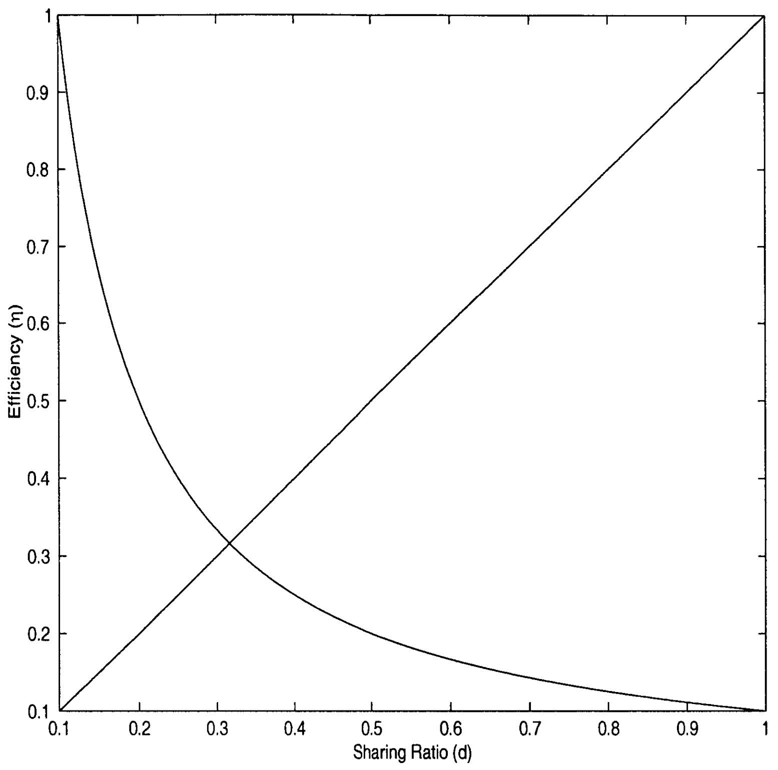


Figure 2-7: Efficiency of a Combined Approach

Chapter 3

The CRAM Design

The previous chapter, investigated how to distribute computing across a network of embedded sensors. It also proposes a model for distributed computing in sensor networks based on local reasoning about a global application. It was argued that with a combination of coarse and fine grained event routing, it was possible to always ensure efficient distribution of computation between co-operating nodes. This chapter describes the structures used to implement the CRAM architecture, and explains how these structures operate.

3.1 Overview

As described in chapter 2, any item of data that enters the system is classified as an event. Each event can be specified by 4 parameters:

1. **Data state:** This describes the current state that the data is in, i.e. the result of the last transformation. For example, an item of data might have a state of "filtered" after environmental noise is removed, or "theft" if it is notification of a theft.

2. **Sensor type:** This describes the type of sensor that the data came from. For example, this might be "RFID" or "temperature" if the data is from a RFID

tag or a temperature reading respectively. This classification is necessary as a single node might control a number of different sensors.

3. **Node ID:** This describes the node that the data originated from. This is necessary as different nodes contain the same types of sensors. For example, every node in a network might be equipped with a RF antenna, a temperature sensor, and a pressure sensor.

4. **Data Value:** This is the actual value of the data. In the case of an RFID tag, it will be an EPC, while in the case of an environmental sensor, it might be a temperature or humidity reading.

Using any combination of these values, an application can specify the events that it wants to receive. As is intuitive, filters that under specify the state, sensor type, and reader ID parameters are coarse-grained, as they will match classes of data. As these refer to groups of data, these filters are usually decided upon when the application is programmed. On the other hand, filters that specify individual events are fine-grained. These filters are most often generated on-the-fly based on local events. Although it is conceivable to modify or add new coarse-grain filters during execution, these events will be rare. Consequently, notifications to add or modify these filters can safely be broadcast to all nodes, as these notifications will most often only happen once. Nodes receiving these broadcasts can then appropriately add or modify the coarse-grained filters. Thus, coarse-grained event filters are replicated on every node. As data enters a node, it is matched against the event filters and is redirected to its subscribers.

However, fine-grain filters change very rapidly, possibly on the order of many times a second. This prevents the constant broadcast of these filters as every node would be saturated with these requests even if they did not produce that event. Therefore, an efficient mechanism should be constructed to handle fine-grained event filters. In order to implement an efficient event-filtering mechanism, CRAM uses a distributed index of events, which is indexed by the event's data value. This index is distributed across all the nodes in the network, each of which stores a small portion of the events.

Each node in the network has the same view of this index. When nodes produce an event, a hash of this event is calculated to determine which node is given the responsibility for storing and propagating this event. The event is then sent to this node, which stores it for a short period of time, after which it discards the event. When a node needs to receive an event via a fine-grained event filter, it calculates a hash of the event. In this way, a node needing to receive a fine-grained event can determine which remote node is responsible for the event. This allows efficient queries for fine-grained events as only the node responsible for the event needs to be contacted.

The implementation of CRAM is detailed in two parts. The anatomy of CRAM describes the structure and connections between each of modules in the system. The physiology details the mechanisms that CRAM uses to operate and describes how data flows through the system.

3.2 The CRAM Anatomy

An overview of CRAM can be seen in figure 3-1. CRAM is divided into four parts:

- **Local cache:** This structure temporarily stores the data items (events) that are harnessed from the sensors local to the node. It also stores any events that are produced during the execution of an application.

- **Memory client:** The job of the memory client is to broker remote events to the application. All events, regardless of the subscriptions type, are received through the memory client. Additionally, both coarse and fine grained event filters dispatched by the application go through the memory client.

- **Memory server:** The memory server oversees the administration of the distributed index. Requests for data that are in the index are handled through the memory server. Additionally, any event produced locally by the node has to be entered into the index. The memory server handles this transfer.

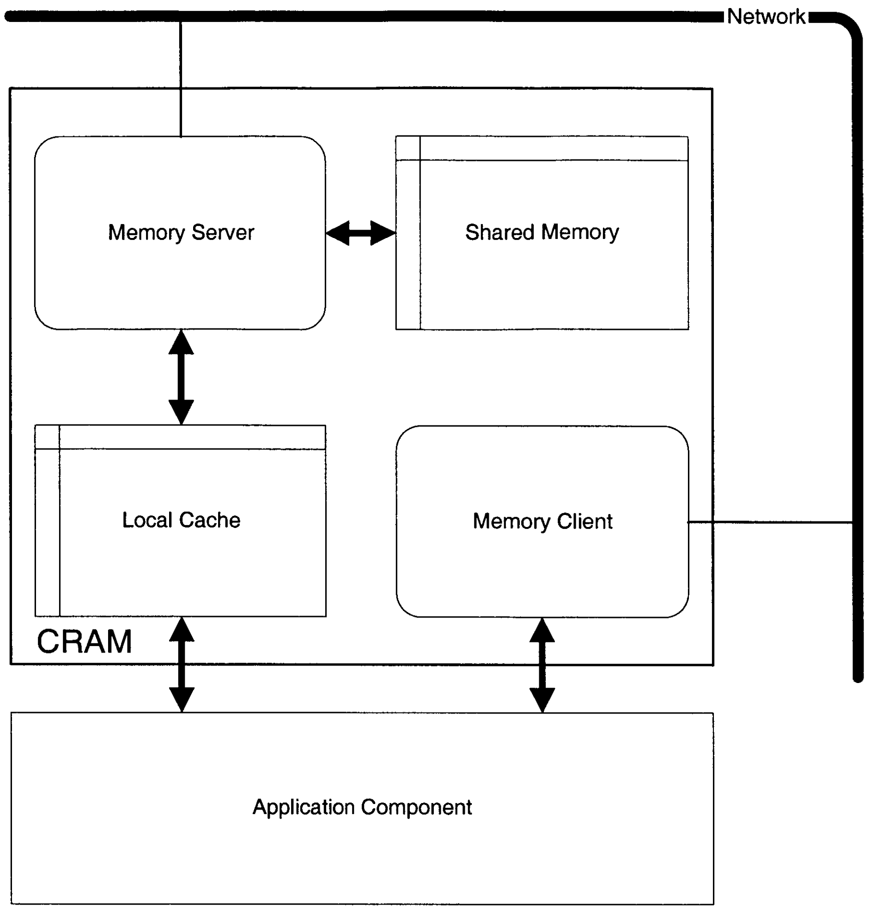


Figure 3-1: The CRAM Anatomy

- **Shared Memory:** This structure holds the portion of the distributed index that is assigned to the node.

These four sections are examined in detail below.

3.2.1 Local Cache

The main function of the local cache is to hold all events produced by the node. This includes events read from the sensors as well as events produced by the application.

As the local cache reads events from the sensors, it is responsible for interfacing with the reader hardware. Moreover, data from the sensors are not tagged i.e. it contains only the data value. The local cache must tag these items with the node ID, sensor type, and data state. The local cache does not tag data received from applications, as it is assumed that the application has already tagged the data.

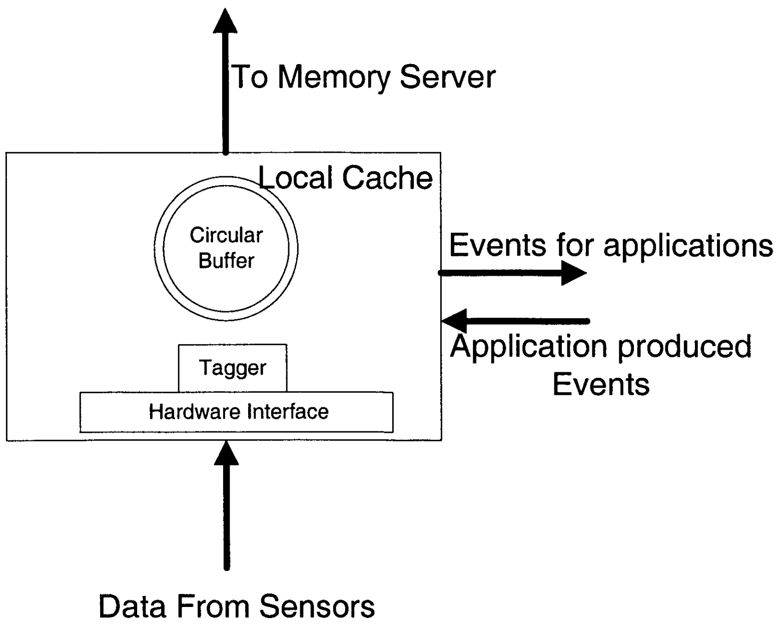


Figure 3-2: The Local Cache

The local cache is implemented as a circular buffer. As events enter the local cache, they are placed in this buffer. As soon as events appear in the buffer, the application components running on the node have access to them. Applications are responsible for keeping track of the last event read. In addition, applications are not allowed to directly modify events in this buffer, as these events still need to be indexed.

The last function of the the local cache is to interface with the memory server. This allows events produced by sensors or applications to be indexed in the distributed index. Although events from applications could go directly to the memory server, this design specifies that they must only go through the local cache. This ensures that applications residing on the same node can receive and send events among themselves without having to go out on the network.

3.2.2 Memory Client

The other module of CRAM that interacts with the application is the memory client. The memory client interacts with the application in two ways:

1. It is responsible for passing all events that the node receives to the components that have requested them

2. It processes event filters from the applications and sends them to the correct nodes. These can be either coarse grain or fine grain filters.

All events that are directed to a node are received directly by the memory client. As there are many application components operating on a single node, each event is tagged with the component that has subscribed to the event. The memory client passes the event only to the subscriber component to prevent overloading the other components operating on the node.

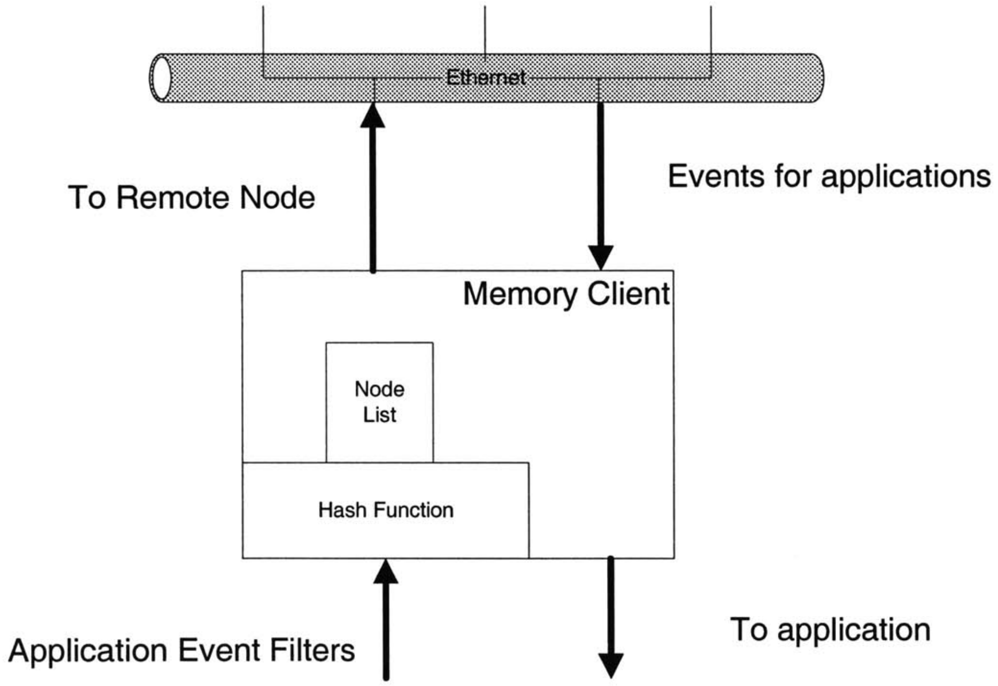


Figure 3-3: The Memory Client

When a component wants to add or modify an event filter, it sends the filter to the memory client. The job of the memory client is to send this filter along to the correct nodes in the network. In the case of coarse grain filters, the memory client simply sends it to all of the nodes. However, in the case of a fine grained filter, the node needs to figure out the specific node. To achieve this, the node uses the hash calculator along with a node list to determine which node to send the fine grained filter to.

3.2.3 Memory Server

The memory server has three functions:

1. Handle storing events in the distributed index.
2. Accept filters from subscribing nodes, and distribute local events to these nodes when appropriate.
3. Distribute coarse-grained locally produced events to the nodes that subscribe to them.

The memory server manages the portion of the distributed index that is hosted on a node. As requests to store or retrieve events come to the node, the memory server receives these requests, and passes them to the shared memory module.

The memory server also handles events that are locally generated. It receives local events from the local cache and sends them along to the node responsible for storing the event. Similar to the memory client, the memory server uses a hash function and a node list to find the node responsible for an event. In addition to inserting events into the event index, the memory server is also responsible for distributing events from the local cache to nodes that have registered for events. For each event that the memory server receives from the local cache, the memory server compares that event with the event filters it has stored, and then distributes the events to the required nodes.

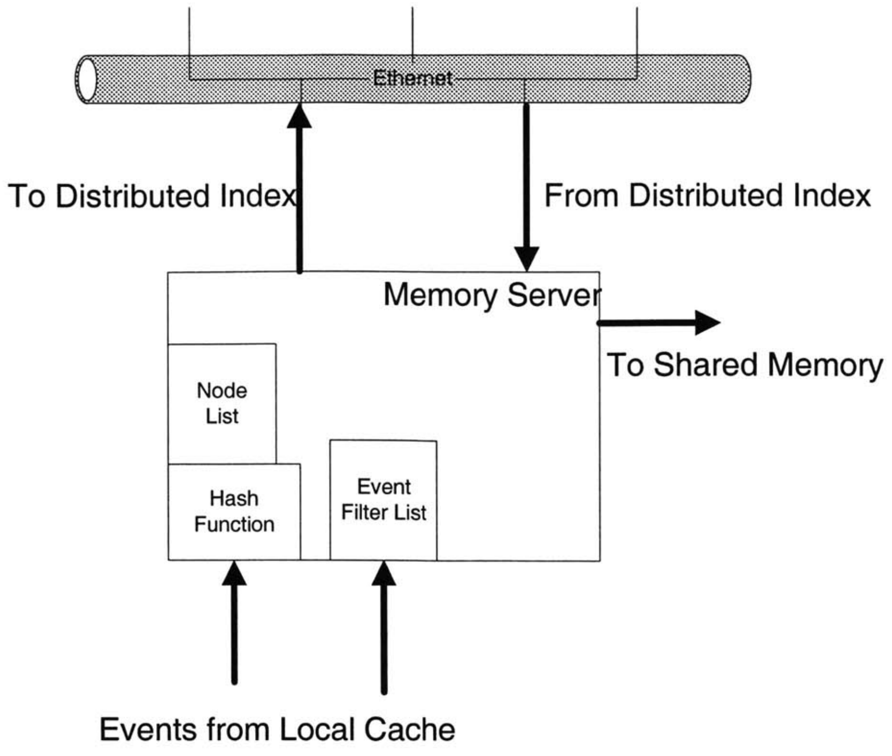


Figure 3-4: The Memory Server

3.2.4 Shared Memory

The shared memory module is the portion of CRAM on which the events are actually stored. The shared memory is implemented as a hash table, where each element in the table is indexed by the data value. This allows events to be quickly retrieved using their data value, which is required for fine grained event filtering. Each event is stored only for a short period of time, after which it is flushed from the memory. If a node does not register a fine-grained filter for an event, that event should not remain in the index. A thread in the shared memory loops through the events discarding them after a specified period of time. This time period is calculated based on the latency of the network and the characteristics of the sensors in the network. This time period has to be enough for the sensor to get a stable reading and transfer that event across the network. In most cases, this period will range from 1-10 seconds.

3.3 The CRAM Physiology

While the previous section described the structures and modules that CRAM is composed of, this section concentrates on mechanisms and data flow.

3.3.1 Networking Protocols

Prior to an examination of CRAM's mechanisms, the network protocols used in CRAM must be briefly discussed. When designing the system, many options were evaluated. At the high end, protocols such as SOAP [28] and XML RPC [31] are possible options. At the low end, there are simple datagrams, while in the middle are various TCP based protocols. However, it was decided to implement all networks communication using simple UDP datagrams. In the CRAM framework, efficiency of communication is of paramount importance. In this arena, UDP protocols have the least overhead. This was especially important as a typical node makes many connections, each of which only transfers a small amount of data. Additionally CRAM uses multicast to send events to multiple recipients. These, coupled with the fact that all communication occurs on a local area network make datagrams a prime choice. For some functions, such as registering coarse grain filters, the use of SOAP or XML RPC was under consideration as these are protocols that make for easier implementation and maintenance. However, in the interest of simplicity, only datagrams are used in CRAM.

3.3.2 Data Flow in a Coarse grained Application

Registering filters

Decisions about which filters to use are made when the programmer develops the application. Consequently, the filters are hard coded in to the application at compile time. When an application starts executing, it joins the coarse-grain filter multicast group (this group is currently designated as 224.100.0.1). It sends a list of all its filters to this group. These filters are then received by the memory server of each

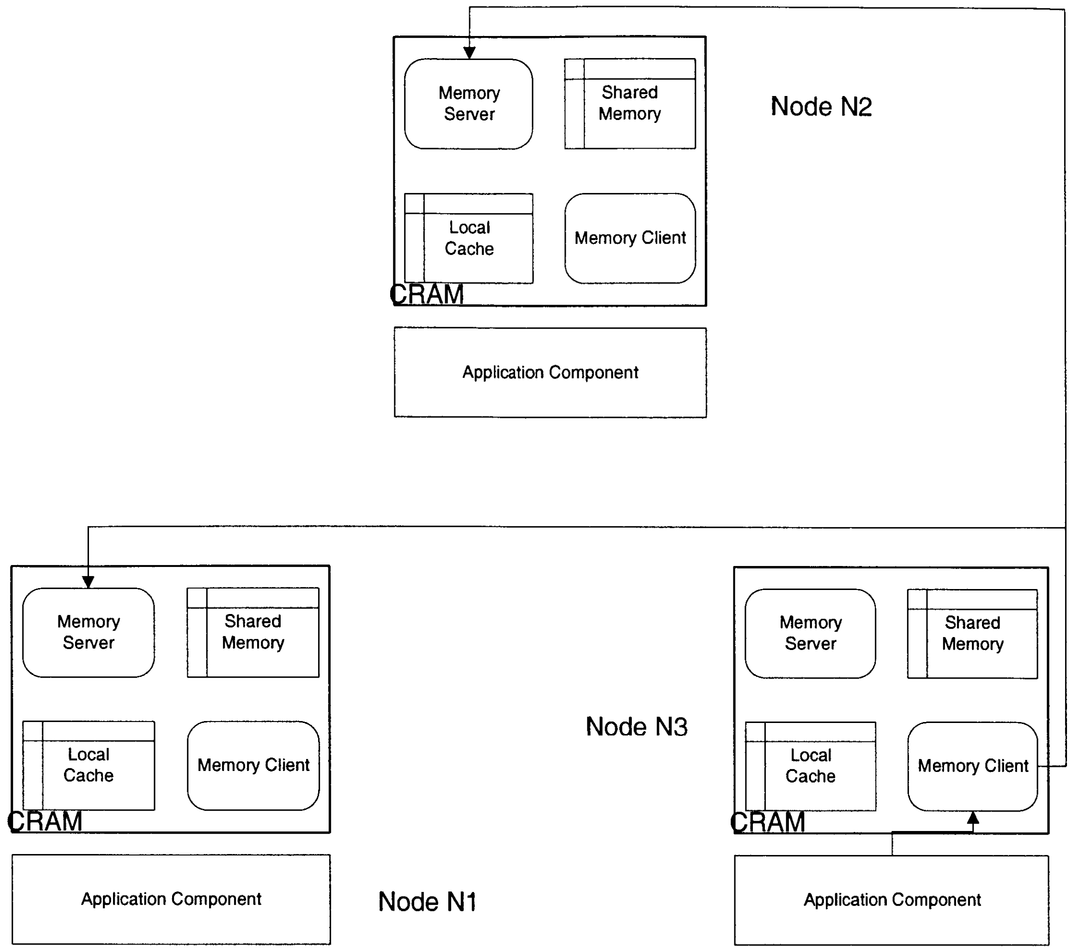


Figure 3-5: Registering of Coarse Event Filters

node and stored in the filter list. This process is shown in figure 3-5. In this network, node N3 sends its filters to the other nodes in the network, N1 and N2.

Sending and Receiving events

The flow of an event from its source to the nodes that subscribes to it is presented in figure 3-6. In this example, node N3 has already registered for the event as described above. A sensor connected to N1 generates the event. The event flows from the sensor into the local cache of N1. Here, it is tagged by the event tagger. Once it is tagged, it is compared to the currently active event filters. As N3 has registered to receive this event, the event is sent to the memory client of N3. The memory client on N3 receives the event and passes it along to the application that registered the event

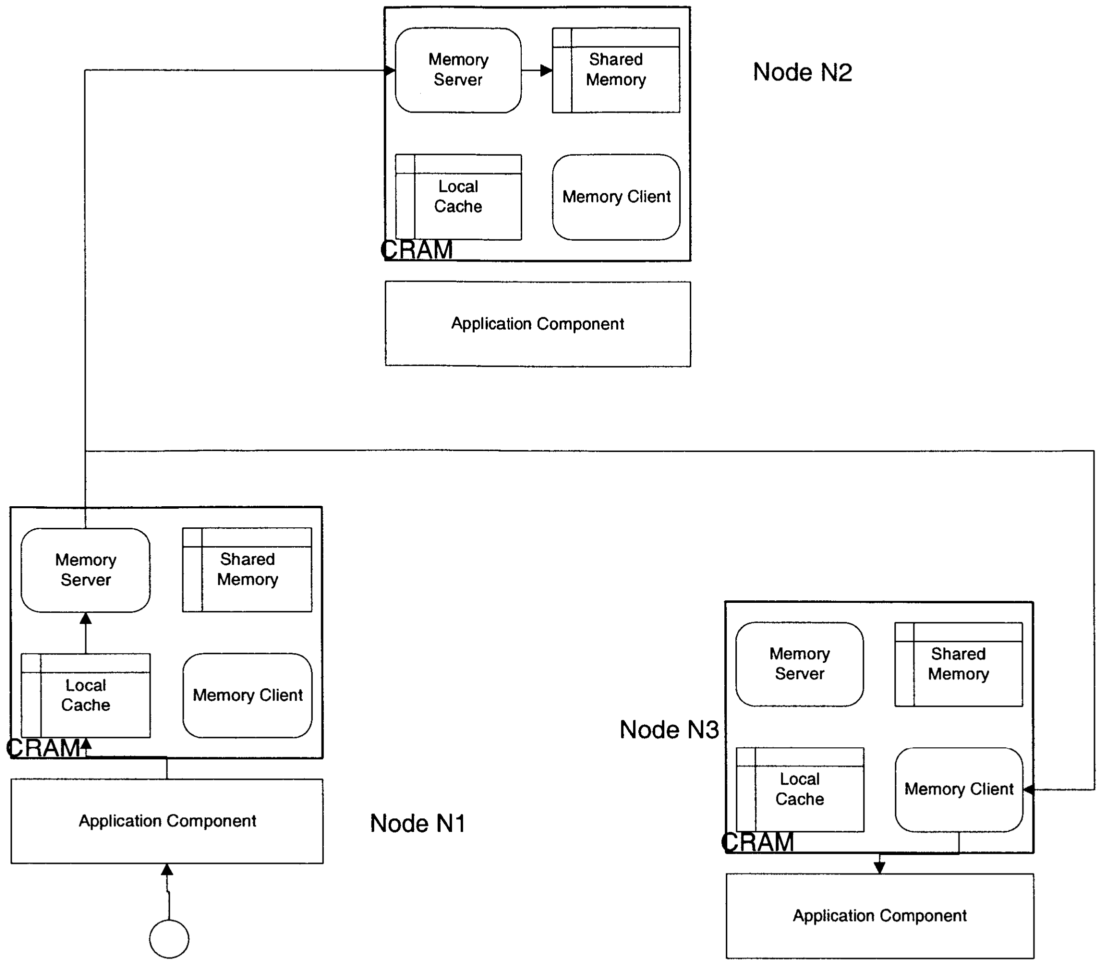


Figure 3-6: Data Flow of Coarse Events

filter. At the same time that the event is matched against the filters, the event also passes through the hash function. The output of the hash function, in conjunction with the node list, is used to determine which node is responsible for hosting the event for the distributed event index. Once the responsible node is determined, the event is sent to the memory server of that node, which in turn stores the event in the shared memory.

3.3.3 Data flow in a Fine grained Application

The data flow for fine-grained routing is depicted in figure 3-7. As both registering and receiving of events take place over very short time scales, they are presented together in one figure, rather than dividing them as in the previous section.

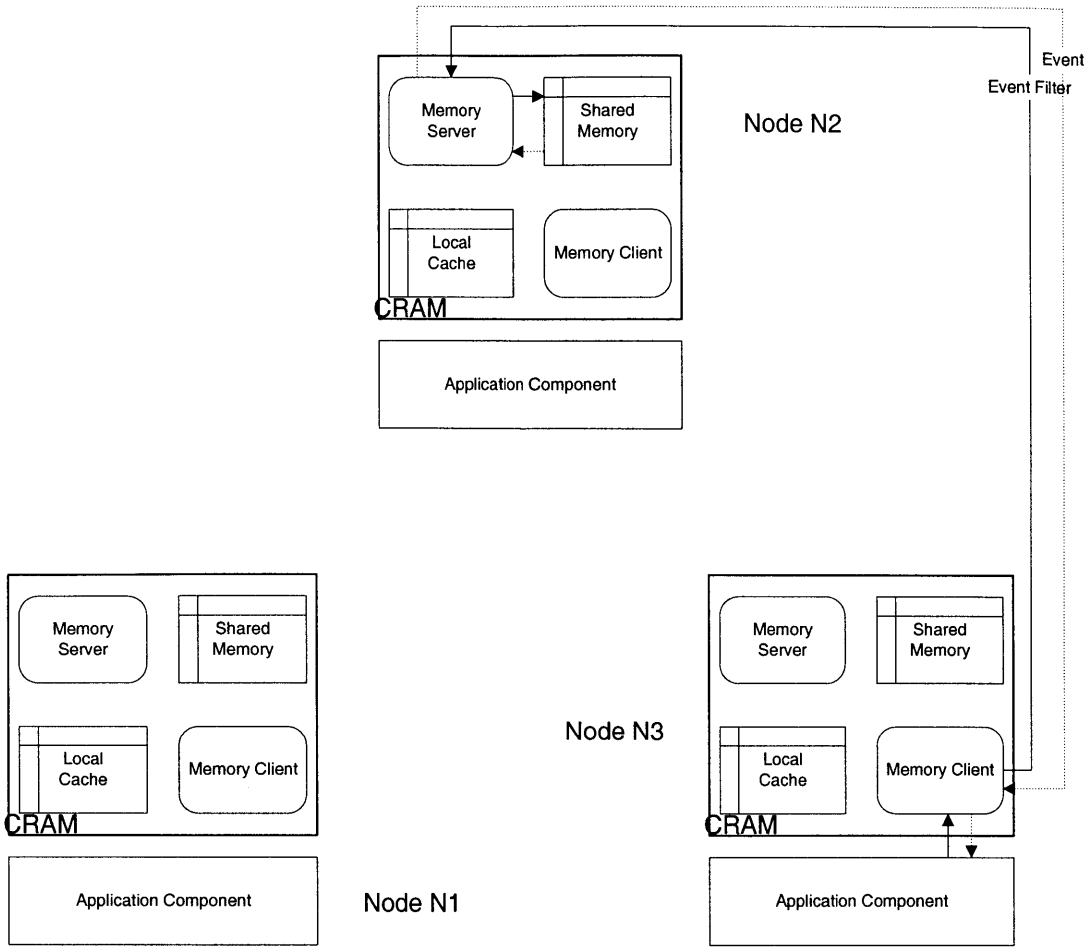


Figure 3-7: Data Flow For Fine-grained Events

In the current scenario, node N1 has produced an event. There are no filters registered, so the event is only entered into the distributed index. As shown in the figure 3-6, the node responsible for this event is N2, and the event is stored in the shared memory at that node.

The component on node N3 is using fine-grained event routing to receive that event. In this case, the application sends the event filter to the memory client. The memory client computes the hash of this filter, and along with the node list identifies the node which would host this event if it occurred. The memory client sends the request for the event to the memory server on that node. The memory server checks the shared memory. If the memory contains the event, it returns it to the memory client on the node that originated the request, otherwise it sends back a null event.

The memory client returns the result to the application.

3.3.4 Node lists

As described, CRAM makes use of node lists, which contain the addresses of all nodes in the network. CRAM is intended for use on networks of at most a few hundred nodes connected on a local area network. Additionally, the rate of nodes joining or leaving the network is very low. Consequently, it is practical for nodes to keep complete lists of all other nodes in the network.

When a node joins a network, it joins a set multicast [currently 224.100.0.2] group. By joining and querying the group, the node makes its presence known and receives information on all currently active hosts in the network. Each node arranges its list (including itself) in ascending order. Consequently, all nodes in the network will have a synchronized view of all the hosts in the network.

If any node detects that another node is not responding for a prolonged period, it sends a message to the multicast group saying that a particular node has left the network. On hearing this, nodes delete the missing node from their list.

Although there are more efficient methods to construct a node list, this design was chosen for its simplicity. If it is necessary for the system to support a very high number of node arrivals and departures, or network partitions, these methods can be employed for the node list. Along with updating node lists, the distributed index has to be updated to account for nodes that leave the network. This is described in section 3.3.5

3.3.5 Hashing and Indexing

This chapter has described how CRAM uses a distributed index to provide support for fine-grained event routing. To ensure a robust, efficient index, this distributed index had to be constructed according to the following constraints:

- The index had to be uniformly addressable from any device in the network.

- The index had to addressable such that it was easy for different applications to request the specific data required by them.
- The arrival or departure of devices would not greatly affect the index
- The memory requirements for the index had to distributed equitably among all nodes.

In order to fulfill these constraints, consistent hashing [16] was used. Consistent hashing is a technique to hash objects across multiple nodes. Consistent hashing was first used for the creation of distributed caches for the web [13]. Recently it has been used in peer-to-peer routing protocols such as Chord[29] and distributed file systems such as CFS [4].

Consistent hashing has two important properties that make it attractive for use in our index. In a network with K keys, and N nodes, with high probability:

1. Each node is responsible for at most $(1 + e)K/N$ keys, where e can be proven to have a bound of $O(logN)$ [16].
2. When a node joins or leaves the network, responsibility for O(K/N) keys change hands, only to or from the joining or leaving nodes [16].

Although these properties hold only under the use of a k-universal hash function, Stoica et al show that the SHA-1 hash function can be used as a good approximation [29].

Consequently, in CRAM, the SHA-1 hash function is used for all hash computations. In conjunction with uniform synchronized node lists, CRAM is able to easily identify which node is responsible for a fine-grained event.

Node Departures and Joins

When a node joins or leaves the network, it is necessary to shift the indexed elements to account for the change in configuration. In the target applications for CRAM, node joins and departures will be relatively infrequent. Node joins will occur when new nodes are added to the network to support new sensors. Node departures will generally occur due to failure of the node hardware. In addition to this, consistent hashing dictates that the number of keys that have to be shifted will be small. Consequently, the current design of CRAM does not optimize this mechanism. If a node joins, each element in the index is rehashed and transferred as necessary. If a node failure is detected, the elements are rehashed similarly to a node join. However, no attempt is made to recover the lost items as fine-grained events are transient in nature. This could lead to some short term (on the order of a few seconds) anomalies in the behavior of the system, but this will be rectified as new events enter the index. This mechanism is intentionally simple due to the uses envisioned for CRAM. However, if the operating parameters become stricter, this mechanism can easily be replaced by more efficient and robust algorithms, such as those presented in the Chord system [29].

3.3.6 Optimizations

The design laid out in this section is a fully implementable version of CRAM. However, this chapter would not be complete without a discussion of possible optimizations and other issues.

Provisions for Transient Hosts

Although the CRAM system is intended to run on fixed hosts on a local area network, it is possible that the operating environments for the system will include a number of transient or mobile hosts. For example, in a warehouse or supermarket, managers or supervisors might carry handheld computers or personal digital assistants (PDAs). For example, a supervisor might want to be informed of theft in progress or a need to restock shelves. In this case, these events can be routed directly to their handheld computer. As these hosts are transient, and generally do not produce events of their own, they are not expected to participate in the routing and storing of events. However, these devices can be given the ability to receive events by implementing a subset of CRAM. In most cases, by implementing the memory client, these devices will be able to receive both coarse and fine grained events.

Provisions for Heterogeneous Network Nodes

In the current design of CRAM, it is assumed that all nodes have roughly equivalent processing power and memory available. Therefore, each node is responsible for storing and routing approximately the same number of events. This assumption is based on the fact that the nodes in the Auto-ID center are RFID tag readers, which for a single local network will be equivalent. However, in other scenarios, the nodes might be more heterogeneous and require responsibility for events in proportion to their resources. There are a number of solutions to allow for this - One simple method is to modify the structure of the node list. When registering with other nodes, a node entering the system also sends along an integer denoting the value of its resources relative to some fixed baseline. For example if the baseline system is 100 MHz with 8 MB RAM, a 200 MHz node with 16MB of RAM would have a resource value of 2, as it has double the processing power and memory. Other nodes would list the address of this node twice in their node lists so as to give this node twice the chance of being picked as a node that has a resource value of 1 (and therefore is listed only once).

Source Quenching

Some event-based routing systems such as Elvin4 [23] implement quenching mechanisms so that only events that are needed are sent into the network. These systems require such a mechanism as all events go through an intermediary server. Therefore, even events that are not needed by any node are sent to the server. Although CRAM does not rely on a central server to relay events, each event is entered into the distributed index in order to support fine-grained event filters. As fine-grained event filters are dynamically generated, usually based on the current data that a node receives, it is impossible to predetermine exactly which events are going to be needed. However, it is still possible to reduce the number of events sent across the network by a sizeable amount. In order to to this, it is possible to require each application to register ‘partial’ fine-grained event filters on the start of their execution. For example, imagine a network of RFID readers, which produce two types of events - raw RFID tag values, and noise-filtered RFID tag values. There is another application in the network that needs to use fine-grained event routing to specify individual RFID tag values. However, it only needs to examine noise filtered tag values. This application can then register a partial filter which corresponds to noise-filtered tag values. Consequently, no raw tag value events will be entered into the index, only noise-filtered events. This mechanism can greatly reduce the number of events in the system if necessary.

Chapter 4

Building Applications with CRAM

This chapter describes how applications are constructed on top of CRAM. Four applications are detailed in order to give a sense of the broad range of applications that can be constructed on top of CRAM. Although there are many possible implementations for these applications, the solutions presented are constructed to illustrate the important characteristics and properties of CRAM. The setting for these applications is a supermarket. The supermarket is modeled as a grid of nodes, each of which has an antenna to read RFID tags. Among the tasks that the system has to perform are:

1. Sending alerts to restock inventory when the inventory is depleted.
2. Sending theft alerts when a theft is detected.
3. Detecting duplicate(counterfeit) tags

4.1 Removing Environmental Noise from the Data: An Example of Coarse Grain Event Routing

The stream of tag IDs read by the RFID readers often contain artifacts due to environmental noise. Before this data is used by any application, the noise needs to be removed. This section details an application, built using CRAM, that can perform this cleanup.

Due to interference, an RFID reader might not be able to read all the tag in its vicinity. Consequently, a tag might not show up during a snapshot even if it is in the field of the reader. This gives the illusion that the tag is not in the field of the reader. As a result of this interference, it appears that a tag is repeatedly leaving (when interference starts) and entering the field of the reader(when interference stops). Since it is not desirable for these anomalies to propagate to applications trying to use this data, they need to be filtered.

The only sure method to detect if a tag has really moved from the field of a reader (as opposed to being obscured by interference), is to check if the tag reappears in the field of a nearby reader. For example, if a tagged product is removed from a shelf, it will show in the field of other readers as a customer walks down the aisles of the supermarket. This property is used to decide if a tag is obscured, or if it has really moved. The application is modeled as an interaction between a node and its neighboring nodes. To receive data from its neighboring nodes, each components sends a filters to the memory client that match all raw RFID data from neighboring nodes. One filter is sent out for each reader that the component needs to couple with. As these are coarse-grained filters, the memory client broadcasts this filter to all active nodes in the network. The memory servers receive these filters and start matching local events to this filter. If events are matched, they are sent directly to the subscribing component.

The component first generates a list of tags that have disappeared in the current snapshot. It compares this list to the remote data (the tags that are present in the current snapshot of its neighbors). If there is a match, then the tag has really left the field of the reader, otherwise it is probably obscured. This application is replicated on all the nodes in the network. Each node registers to receive all the raw data from its neighboring nodes, which is a coarse-grained filter. As nodes produce these events, they are routed to the correct nodes. The node can then decide whether they are obscured or whether they have left the field of the reader.

Table 4.1 gives a summary of this application. It is important to note that this application uses coarse-grain filtering. Each component is coupled with its neighbor-

Component Name	Noise Reducing Filter
Located At	All nodes
Coupled With	Neighboring nodes
Event Filters $[state, sensor, nodeID, datavalue]$	[raw, RFID, x, *] Where x is all neighboring nodes
Events Produced	$[Filtered, RFID, readerID, EPC]$

Table 4.1: Summary of Noise Reducing Filter Application

ing components. For example, if each node is coupled with four neighbors, and the scope of this application is 100 nodes, then $d = 0.04$. As this is a small coupling-to-scope ratio, coarse-grain routing was chosen for this application. The efficiency of this application can be determined from figure 2-7 in section 2.5.4. As we can see, although this application does not have the same efficiency as a centralized version of this application, the efficiency is still extremely good.

4.2 Applications with Heterogeneous Sub-Components: Theft Detection

The noise-reduction application presented above is an example of an application with homogeneous components: the application is modeled as a single local-remote application, and this component is replicated across multiple nodes. However, applications can consist of heterogeneous components spread across different nodes. This section presents a theft detection application built with heterogeneous components.

The theft-detection applications works by comparing all products that go through the exits with all those which have been paid for. Products that leave the supermarket are detected by RFID tag readers at the exits, while products that have been paid for are detected by a reader at checkout. The application consists of two different components. The first component resides on the readers at the checkout lines. It produces events when products have been paid for, but does not subscribe to any events. The second component resides on the exit readers. Its interaction is with events produced by the exit readers. These components register coarse grain filters to receive all 'sold' events from the cash register and keeps a list of them. If any product enters its field that is not on its sold list, it sends out a theft event. As tags pass though its field that have already been paid for, the components send out 'exited' events, which are subscribed to by all exit readers. On receiving these events, the exit components can remove the products from their list.

Component Name	Purchase Checker	Exit Checker
Located At	All cash registers	All exits
Coupled With	None	Exits and registers
Event Filters $[state, sensor, nodeID, datavalue]$	none	$[sold, *, *, *]$
Events Produced	$[sold, RFID, readerID, EPC]$	$[theft, RFID, readerID, EPC]$ $[theft, RFID, readerID, EPC]$

Table 4.2: Summary of Theft Detecting Application

4.3 Applications Can Built Upon Each Other: Inventory Control

An important feature of CRAM is that applications can build on top of events created by totally different applications. Although most applications in a RFID environment will often implicitly build on events from a noise-reduction application such as presented in Section 4.1, another example is detailed here as it shows how multiple applications can use intermediate event results in different ways. This section shows how an inventory application can build on the events generated by the theft detection application. In a RFID tag enabled supermarket, tag readers can detect when the shelf needs to be restocked. However, products can be moved around by customers, or remain their shopping cart. Consequently, inventory records should only be depleted when a product actually leaves the building. In order to achieve this, an inventory control application would subscribe to 'exited' events from the exit readers, signaling that a product has left the building[1]. On receiving these events, the inventory component can deduct appropriately from the inventory record. This method of building

[1]Another strategy would be to subscribe to 'sold' events from cash register components. However, in this case 'exited' events are subscribed to in order reduce the occurrence of sold inventory being returned

upon events from different applications is a core feature of transformational routing. It saves the developer from having to develop identically functional components for application. More importantly, it greatly reduces the amount of traffic sent around the network, as events used to compute intermediate results are not replicated - the intermediate results are used directly.

Component Name	Inventory Control
Located At	Inventory Databases
Coupled With	None
Event Filters $[state, sensor, nodeID, datavalue]$	[exit, RFID, *, *]
Events Produced	none

Table 4.3: Summary of Noise Reducing Filter Application

4.4 Fine grained event routing: Detecting Duplicate Tags

In the Auto-ID system, each product is tagged with an RFID tag that contains a 96-bit Electronic Product Code (EPC). Since the EPC on every item is unique, no two tags interrogated by a reader should return the same EPC value. The detection of duplicate EPCs signals a reader error or possible counterfeit tags.

The purpose of this application is to ensure that if duplicate EPCs are detected by the system, they are recognized as such and an error is signaled. To perform this task, a reader must check if any duplicates exist in the locally read tags. Additionally, it must compare its local data to the data from all the other readers to ensure that no duplicates exist. For this application, every component will need to be coupled with all the nodes in the system. This is because a locally generated EPC event needs to be compared with all others EPC events in the network. Consequently, this application is a prime candidate for fine-grained event routing.

Each component operates as follows. First, it reads all locally generated EPC events and ensures that no duplicates exist in this set. Next, it sends out a fine-grained event filter that matches the EPC that has just been read. When the fine-grained filter is sent out to the application, it goes first to the memory client. The memory client calculates the hash of this filter to compute the node responsible for this event. It then sends this query directly to that node. If the node does not have a match for the event in its shared memory, then no reader in the network has detected a tag with the that EPC. This is because if any reader had generated that EPC, it would have been hashed and stored on same node that the filter went to. If there is a match in the shared memory, this event is returned, which indicates that a duplicate tag has been detected. However, there is one caveat. It is possible that the radio fields of neighboring readers overlap. In this case, two readers might detect the same EPC value. However, this is not a duplicate EPC because both readers are actually reading the same tag. To account for this, if a match is returned, the component checks the event to determine which reader it has come from. If the reader is a neighboring one, then this match is discarded. Once a component is sure that there is a duplicate EPC, it generates a 'duplicate' event to signal this occurrence.

Component Name	Duplicate Detector
Located At	All nodes
Coupled With	All nodes
Event Filters $[state, sensor, nodeID, datavalue]$	[*, *, *, x] Where x is the locally generated EPC
Events Produced	$[Duplicate, RFID, readerID, EPC]$

Table 4.4: Summary of Noise Reducing Filter Application

Table 4.4 gives a summary of this application. As the filters sent out by this application refer to a specific EPC value, which is dynamically generated from local events, this application uses fine-grained routing. In order to to function correctly, each component needs to be coupled with every other node.Consequently, $d =$ for this application. As this is the highest possible coupling-to-scope ratio, fine-grained routing was chosen for this application. The efficiency of this application can be determined from figure 2-7 in section 2.5.4. As we can see, by using fine-grained routing in this situation, this application achieves a 100% efficiency compared to a centralized version of this application.

Chapter 5

Conclusions and Future Work

5.1 Discussion

In this thesis I have presented a framework for decentralized, distributed processing in embedded sensor networks. This framework is built on a model of reasoning locally about a global application. Components that build from local-remote interactions are then replicated across multiple nodes. The resulting combination of components give rise to globally relevant results.

The CRAM design was prompted by a need for an incrementally scalable system that could process data from networked embedded sensors without the need for centralized resources. CRAM fills an increasingly important area in embedded systems. Many systems for controlling embedded sensors are either centralized. Others, which promote a distributed approach to control, are directed towards the extreme low end of the spectrum of embedded systems - sensors with extremely limited processing power and connectivity [12]. CRAM bridges the gap between these approaches by targeting a range of systems that are increasing deployed in industrial and home automation systems. In these systems, a node connected on a local area network controls a small number of sensors. However, the entire network may be composed of hundreds of these nodes. CRAM leverages the flexibility offered by this increased processing power and connectivity to provide a platform upon which sophisticated global applications can be constructed. As applications are built with local reasoning, the programmer needs only to have a vague understanding of how the entire network operates. The system appears to the programmer simply as a single node interacting with a cloud of remote nodes. As embedded networks grow in size and complexity, this is increasingly important as it is impossible to understand every device in the network.

While designing CRAM, the notion of a sensor underwent a considerable transformation. At the onset, the notion of a sensor consisted only of environmental sensors such as temperature and humidity, or identification sensors such as RFID devices. However, this notion has expanded considerably to include a wide variety of devices. For example, one does not usually think of a video camera as a sensor in the conventional sense of the word. However, one can imagine using CRAM to service security and monitoring applications in which multiple cameras are used over a wide area. A processor connected to each camera would be perform local image processing tasks, but would able to harness data from multiple sources (cameras and other sensors) to augment its results.

5.2 Integrating CRAM into the Auto-ID Architecture

As explained in section 1.4, the Auto-ID architecture is hierarchical. At the lowest layer, are the RFID readers that read products tagged with EPCs. In the current incarnation of the system, the readers do not perform any processing[1]. On reading a tag, the readers send this unprocessed data to the *leaf-level savant.* The savant is a server, whose responsibility it is to control all the readers in a given area, process this data, and send it along to the next highest level. For example, a leaf-level savant controls all the readers in a warehouse and processes all the data from these readers. It is responsible for all low-level applications such as theft detection and restocking alerts. It also sends aggregate data (e.g daily totals) to the a regional savant. CRAM

[1]Most readers perform some form of tag anti-collision [26]. However, as this is usually tightly integrated with the reader hardware it is considered part of the tag interrogation process

was motivated by problems anticipated with having a single leaf-level savant perform the processing for hundreds of readers.

The CRAM is designed to replace the need for a leaf-level savant. Instead of having a single savant process the data, the processing will be performed co-operating readers. This is possible as the current readers have ample computing power to process the individual data that they generate. Although the leaf-level savant is designed to operate very differently from CRAM, it is possible to modify and scale down the savant design to produce a *distributed savant* based on the CRAM architecture. This is important as CRAM will not be have to built from the ground up. First the savant incorporates a task manager to allow tasks to be remotely dispatched to it. This mechanism can be modified to support the downloading of components onto the reader. The savant also incorporates a primitive coarse-grain routing mechanism, which can be adapted to perform the coarse-grain routing described in this thesis. Additionally, the savant caches all events that are sent to it in a circular buffer. This cache can be modified into the local cache described in CRAM. The fine-grained event routing mechanism will have to be developed from scratch, as no equivalent exists in the leaf-level savant. However, this is not anticipated to be a very hard task as many implementations of indexes based on consistent hashing are already in existence [29, 4]. Additionally, a CRAM compliant version of the savant would have to integrate directly with the reader hardware. The current focus of the savant architecture is based on handling a high number of transactions per second. In a distributed savant, however, the focus would need to be on memory and communication efficiency as each mini savant would be replicated on each reader.

5.3 Future Work

While this thesis has designed and developed a framework, there is scope for further research:

- **Formal Modeling:** Although CRAM provides mechanisms for performing both coarse and fine-grained event routing, a formal analysis how often each of these will be used has not been performed. As the communication mechanism is dependent on the application, this would involve analyzing multiple applications to understand what fraction of applications are loosely coupled and what fraction are tightly coupled.

- **Simulations:** While an understanding of how CRAM performs for single applications has been achieved , more work is required to examine how CRAM scales as multiple applications are added to the system. This would probably be best performed by simulating multiple applications on top of the CRAM system. An examination in this area would probably also lead to many optimizations of the system.

- **Getting components to their destinations:** This thesis, has not discussed how components are loaded onto each node. It was simply stated that components would be replicated across many nodes. Although it is possible to manually load components onto nodes, this is not practical as the number of nodes gets increases. One solution to this problem would be to enable components be able to move across the network. For example, once an application is written, it is loaded onto the network from any point. From there it can travel to the nodes on which it is supposed to operate.

- **Unified Filter Model:** In the current incarnation of CRAM, the burden of deciding to use coarse or fine grained filtering is placed on the programmer. An interesting direction for future research would involve a unified model for filters. Thus, the programmer would not need to differentiate between different types of filters. When the application is compiled, the coupling of the application could be examined, based upon which filters would be instantiated into coarse or fine-grained as needed.

5.4 Conclusion

This thesis has argued for the necessity of a decentralized approach to processing in sensor networks. CRAM was designed to fulfill this requirement. The local reasoning model was developed to support distributed computing in sensor networks. In this model, components residing on multiple nodes process parts of the application in parallel while sharing data using event-based communication. It is shows that by using a combination of coarse-grained and fine-grained routing, CRAM can achieve a computational efficiency that is just a small constant factor less than a centralized system, while still providing the benefits of a decentralized system. This thesis details the design of CRAM and describes how complex applications can be built on top of this architecture.

Bibliography

[1] Brock, D. L. The compact electronic product code - a 64-bit representation of the electronic product code. Tech. Rep. MIT-AUTOID-WH-008, MIT Auto-ID Center, November 2001.

[2] Brock, D. L. The electronic product code (epc)- a naming scheme for physical objects. Tech. Rep. MIT-AUTOID-WH-002, MIT Auto-ID Center, January 2001.

[3] Brock, D. L., Milne, T. P., Kang, Y. Y., and Lewis, B. The physical markup language. Tech. Rep. MIT-AUTOID-WH-005, MIT Auto-ID Center, June 2001.

[4] Dabek, F., Kaashhoek, M., Karger, D., Morris, R., and Stoica, I. Wide-area cooperative storage with cfs. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)* (2001).

[5] Estrin, D., Govindan, R., and Heidemann, J. Embedding the internet: introduction. *Communications of the ACM 43*, 5 (2000), 38–41.

[6] Foley, J. T. An infrastructure for electromechanical appliances on the intrnet. Master's thesis, Massachusetts Institute of Technology, 1999.

[7] Foster, I., and Kesselman, C. The globus project: A status report, 1998.

[8] Foster, I. T. The anatomy of the grid: Enabling scalable virtual organizations. In *European Conference on Parallel Processing* (2001), pp. 1–4.

[9] GELERNTER, D. Generative communication in linda. In *ACM Transactions on Programming Languages and Systems* (January 1985).

[10] GELERNTER, D., CARRIERO, N., CHANDRAN, S., AND CHANG, S. Parallel programming in linda. In *International Conference on Parallel Processing* (August 1985), pp. 255–263.

[11] INTANAGONWIWAT, C., GOVINDAN, R., AND ESTRIN, D. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the sixth annual international conference on Mobile computing and networking* (2000), ACM Press, pp. 56–67.

[12] KAHN, J. M., KATZ, R. H., AND PISTER, K. S. J. Next century challenges: mobile networking for smart dust. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking* (1999), ACM Press, pp. 271–278.

[13] KARGER, D. R., LEHMAN, E., LEIGHTON, F. T., PANIGRAHY, R., LEVINE, M. S., AND LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing* (May 1997), pp. 654–663.

[14] KUBIATOWICZ, J., BINDEL, D., CHEN, Y., EATON, P., GEELS, D., GUMMADI, R., RHEA, S., WEATHERSPOON, H., WEIMER, W., WELLS, C., AND ZHAO, B. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS* (November 2000), ACM.

[15] LEOPOLD, C. *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches.* John Wiley and Sons Inc., 2001.

[16] LEWIN, D. M. Consistent hashing and random trees : algorithms for caching in distributed networks. Master's thesis, Massachusetts Institute of Technology, 1998.

[17] Opyrchal, L., Astley, M., Auerbach, J., Banavar, G., Strom, R., and Sturman, D. Exploiting ip multicast in content-based publish-subscribe systems. In *IFIP/ACM International Conference on Distributed systems platforms* (2000), Springer-Verlag New York, Inc., pp. 185–207.

[18] Pottie, G. J., and Kaiser, W. J. Wireless integrated network sensors. *Communications of the ACM 43*, 5 (2000), 51–58.

[19] Rowstron, A. I. T., and Druschel, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware* (2001), pp. 329–350.

[20] Rowstron, A. I. T., Kermarrec, A.-M., Castro, M., and Druschel, P. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication* (2001), pp. 30–43.

[21] Sarma, S. Towards the 5 cent tag. Tech. Rep. MIT-AUTOID-WH-006, MIT Auto-ID Center, November 2001.

[22] Sarma, S., Brock, D. L., and Ashton, K. The networked physical world - proposals for engineering the next generation of computing, commerce and automatic identification. Tech. Rep. MIT-AUTOID-WH-001, MIT Auto-ID Center, October 2000.

[23] Segall, B., Arnold, D., Boot, J., Henderson, M., and Phelps, T. Content based routing with elvin4. In *Proceedings of AUUG2K* (June 2000).

[24] The seach for extra terrestrial intelligence at home (seti@home). http://setiathome.ssl.berkeley.edu/.

[25] Sheldon, M. A. *Content routing : a scalable architecture for network-based information discovery.* PhD thesis, Massachusetts Institute of Technology, 1995.

[26] Siu, K.-Y., Law, C., and Lee, K. Efficient memoryless protocol for tag identification. Tech. Rep. MIT-AUTOID-TR-004, MIT Auto-ID Center, August 2000.

[27] SNOEREN, A. C., CONLEY, K., AND GIFFORD, D. K. Mesh-based content routing using xml. In *Proceedings of the 18th symposium on Proceedings of the 18th ACM symposium on operating systems principles* (2001), ACM Press, pp. 160–173.

[28] Simple object access protocol (soap). http://www.w3c.org/soap.

[29] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications, 2001.

[30] Xmlblaster. http://www.xmlblaster.org.

[31] Xml remote procedure call (xml-rpc). http://www.w3c.org/xmlrpc.

[32] ZHUANG, S., ZHAO, B., JOSEPH, A., KATZ, R., AND KUBIATOWICZ, J. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.