DFuse: A Framework for Distributed Data Fusion *

Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin,
Phillip Hutto, Arnab Paul, and Umakishore Ramachandran

{rajnish, wolenetz, bikash, espress, pwh, arnab, rama}@cc.gatech.edu

College of Computing
Georgia Institute of Technology

ABSTRACT

Simple in-network data aggregation (or fusion) techniques for sensor networks have been the focus of several recent research efforts, but they are insufficient to support advanced fusion applications. We extend these techniques to future sensor networks and ask two related questions: (a) what is the appropriate set of data fusion techniques, and (b) how do we dynamically assign aggregation roles to the nodes of a sensor network ? We have developed an architectural framework, *DFuse*, for answering these two questions. It consists of a data fusion API and a distributed algorithm for energy-aware role assignment. The fusion API enables an application to be specified as a coarse-grained dataflow graph, and eases application development and deployment. The role assignment algorithm maps the graph onto the network, and optimally adapts the mapping at run-time using role migration. Experiments on an iPAQ farm show that the fusion API has low-overhead, and the role assignment algorithm with role migration significantly increases the network lifetime compared to any static assignment.

Categories and Subject Descriptors : D.4.7 [Operating Systems]: Organization and Design – Distributed Systems, and Embedded Systems.

General Terms : Algorithms, Design, Management, Measurement.

Keywords : Sensor Network, In-network aggregation, Data fusion, Role assignment, Energy awareness, Middleware, Platform.

*The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF grant CCR-99-72216, HP/Compaq Cambridge Research Lab, the Yamacraw project of the State of Georgia, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp.

SenSys'03, November 5–7, 2003, Los Angeles, California, USA.

1. INTRODUCTION

With advances in technology, it is becoming increasingly feasible to put a fast processor on a single small sensor along with a sizable memory and a radio transceiver. There is an ever-evolving continuum of sensing, computing, and communication capabilities from smartdust, to sensors, to mobile devices, to desktops, to clusters. With this evolution, capabilities are moving from larger footprint to smaller footprint devices. For example, tomorrow's *mote* will be comparable in resources to today's mobile devices; and tomorrow's mobile devices will be comparable to current desktops. These developments suggest that future sensor networks may well be capable of supporting applications that require resource-rich support today. Examples of such applications include streaming media, surveillance, image-based tracking and interactive vision. Many of these *fusion applications* share a common requirement, namely, hierarchical *data fusion*, i.e., applying a synthesis operation on input streams.

This paper focuses on challenges involved in supporting fusion applications in *wireless ad hoc sensor networks* (WASN). Developing fusion applications is challenging in general because of the time-sensitive nature of the fusion operation, and the need for synchronization of the data from multiple streams. Since the applications are inherently distributed, they are typically implemented via distributed threads that perform fusion in a hierarchical manner. Thus, the application programmer has to deal with thread management, data synchronization, buffer handling, and exceptions (such as time-outs while waiting for input data for a fusion function) - all in a distributed fashion. WASN add another level of complexity to such application development due to the scarcity of power in the individual nodes [5]. In-network aggregation and power-aware routing are techniques to alleviate the power scarcity of WASN. While the good news about fusion applications is that they inherently need in-network aggregation, a naive placement of the fusion functions on the network nodes will diminish the usefulness of in-network fusion, and reduce the longevity of the network (and hence the application). Thus, managing the placement (and dynamic relocation) of the fusion functions on the network nodes with a view to saving power becomes an additional responsibility of the application programmer. Dynamic relocation may be required either because the remaining power level at the current node is going below threshold, or to save the power consumed in the network as a whole by reducing the total data transmission. Supporting the relocation of fusion functions at run-time has all the traditional challenges of process migration [15].

We have developed *DFuse*, an architecture for programming fusion applications. It supports distributed data fusion with automatic management of fusion point placement and migration to optimize a given cost function (such as network longevity). Using the DFuse framework, application programmers need only implement the fusion functions and provide the dataflow graph (the relationships of fusion functions to one another, as shown in Figure 1). The fusion API in the DFuse architecture subsumes issues such as data synchronization and buffer management that are inherent in distributed programming.

The main contributions of this work are summarized below:

1. Fusion API: We design and implement a rich API that affords programming ease for developing complex sensor fusion applications. The API allows any synthesis operation on stream data to be specified as a fusion function, ranging from simple aggregation (such as min, max, sum, or concatenation) to more complex perception tasks (such as analyzing a sequence of video images). This is in contrast to current in-network aggregation approaches [11, 8, 6] that allow only limited types of aggregation operations as fusion functions.

2. Distributed algorithm for fusion function placement and dynamic relocation: There is a combinatorially large number of options for placing the fusion functions in the network. Hence, finding an optimal placement that minimizes communication is difficult. Also, the placement needs to be re-evaluated quite frequently considering the dynamic nature of WASN. We develop a novel heuristic-based algorithm to find a *good* (according to some predefined cost function) mapping of fusion functions to the network nodes. The mapping is re-evaluated periodically to address dynamic changes in nodes' power levels and network behavior.

3. Quantitative evaluation of the DFuse framework: The evaluation includes micro-benchmarks of the primitives provided by the fusion API as well as measurement of the data transport in a tracker application. Using an implementation of the fusion API on a wireless iPAQ farm coupled with an event-driven engine that simulates the WASN, we quantify the ability of the distributed algorithm to increase the longevity of the network with a given power budget of the nodes.

The rest of the paper is structured as follows. Section 2 analyzes fusion application requirements and presents the DFuse architecture. In Section 3, we describe how DFuse supports distributed data fusion. Section 4 explains a heuristic-based distributed algorithm for placing fusion points in the network. This is followed by implementation details of the framework in Section 5 and its evaluation in Section 6. We then compare our work with existing and other ongoing efforts in Section 7, present some directions for future work in Section 8, and conclude in Section 9.

2. DFUSE ARCHITECTURE

This section presents the DFuse architecture. First, we explore target applications and execution environments to identify the architectural requirements. We then describe the architecture and discuss how it is to be used in developing fusion applications.

2.1 Target Applications and Execution Environment

DFuse is suitable for applications that apply hierarchical fusion functions (input to a fusion function may be the output of another fusion function) on time-sequenced data items. A fusion operation may apply a function to a sequence of stream data from a single source, from multiple sources, or from a set of sources and other fusion functions.

DFuse accepts an application as a task graph, where a vertex in the task graph can be one of *data source*, *data sink*, or *fusion point*. A data source represents any data producer, such as a sensor or a standalone application. DFuse assumes that data sources are known at query time (when the user specifies the application task graph). A data sink is an end consumer, including a human in the loop, an application, an actuator, or an output device such as a display. Intermediate fusion points perform application-specific processing on streaming data. Thus, an application is a directed graph, with the data flow (i.e. producer-consumer relationships) indicated by the directionality of the associated edge between any two vertices.

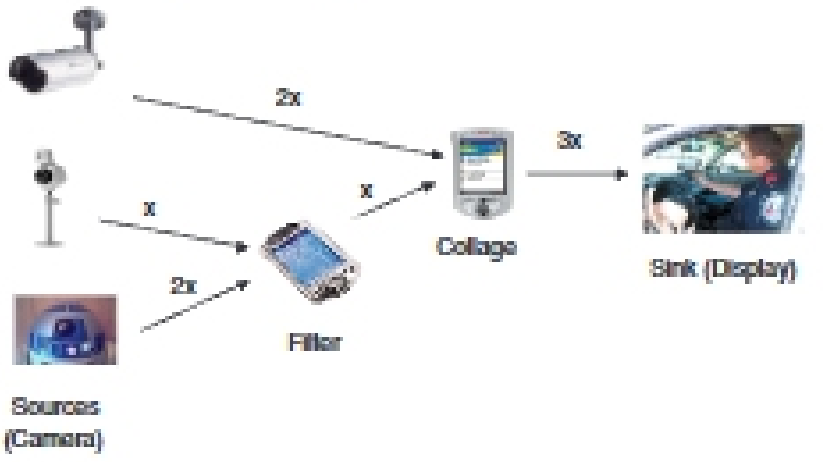

Figure 1: An example tracking application that uses distributed data fusion. Here, *filter* and *collage* are the two fusion points taking inputs from cameras, and the face recognition algorithm is running at the sink. Edge labels indicate relative (expected) transmission rates of data sources and fusion points.

For example, Figure 1 shows a task graph for a tracking application. The filter fusion function selects images with some interesting properties (e.g. rapidly changing scene), and sends the compressed image data to the collage function. Thus, the filter function is an example of a fusion point that does data contraction. The collage function uncompresses the images coming from possibly different locations. It combines these images and sends the composite image to the root (sink) for further processing. Thus, the collage function represents a fusion point that may do data expansion.

DFuse is intended for deployment in a heterogeneous ad hoc sensor network environment. However, DFuse cannot be deployed in current sensor networks given the limited capabilities available in sensor node prototypes such as Berkeley motes [7]. But, as we add devices with more capabilities to the sensor network, or improve the sensor nodes themselves, more demanding applications can be mapped onto such networks and DFuse provides a flexible fusion API for such a deployment. As will become clear in later sections, DFuse handles the dynamic nature of such networks by employing a resource-aware heuristic for placing the fusion points in the network.

DFuse assumes that any node in the network is reachable from any other node. Further, DFuse assumes a routing layer that exposes hop-count information between any two nodes in the network. Typically, such support can be provided by a separate layer that supports a routing protocol for ad hoc networks, like Dynamic Source Routing (DSR) [10], and exposes an interface to query the routing information.

2.2 Architecture Components

Figure 2(A) shows a high-level view of the DFuse architecture that consists of two main runtime components: *fusion module* and *placement module*. The fusion module implements the fusion API used in the development of the application. The fusion module interacts with the placement module to determine a *good* mapping of the fusion functions to the sensor nodes given the dynamic state of the network and the application behavior. These two components constitute the runtime support available in each node of the network.

Figure 2(B) shows the internal structure of the fusion module. Details of the fusion module are discussed in section 3. The modules that implement resource monitoring and routing are external to the DFuse architecture. These modules help in the evaluation of cost functions that is used by the placement module in determining a *good* placement of fusion functions.

Launching an Application and Network Deployment

An application program consists of two entities: a *task graph*, and the code for the *fusion functions* that need to be run on the different nodes of the graph. DFuse automatically generates the glue code for instantiating the task graph on the physical nodes of the network. DFuse also shields the application programmer from deciding the placement of the task graph nodes in the network.

Launching an application is accomplished by presenting the task graph and the fusion codes to DFuse at some designated node, let us call it the *root* node. Upon getting this launch request, the placement module of DFuse at the root node starts a distributed algorithm for determining the best placement (details to be presented in Section 4) of the fusion functions. The algorithm maps the fusion functions of the task graph onto the physical network subject to some cost function. In this resulting overlay network, each node knows the fusion function (if any) it has to run as well as the sources and sinks that are connected to it. The resulting overlay network is a directed graph with source, fusion, and sink nodes (there could be cycles since the application may have feedback control). The application starts up with the sink nodes running their respective codes, resulting in the transitive launching of the codes in the intermediate fusion nodes and eventually the source nodes. Cycles in the overlay network are handled by each node remembering if a launch request has already been sent to the nodes that it is connected to.

The role of each node in the network can change over time due to both the application dynamics as well as health of the nodes. The placement module at each node performs periodic re-evaluation of its health and those of its neighbors to determine if there is a better choice of placement of the fusion functions. The placement module requests the fusion module to affect any needed relocation of fusion functions in the network. Details of the placement module are forthcoming in Section 4.

The fusion module at each node of the network retrieves the fusion function(s) to be launched at this node. It is a space-time trade-off to either retrieve a fusion function on-demand or store the code corresponding to all fusion functions at every node of the network. The latter design choice will enable quick launching of a fusion function at any node while increasing the space need at each node.

3. DISTRIBUTED DATA FUSION SUPPORT

DFuse utilizes a package of high-level abstractions for supporting fusion operations in stream-oriented environments. This package, called *Fusion Channels*, is conceptually language and platform independent.

Data fusion, broadly defined, is the application of an arbitrary transformation to a correlated set of inputs, producing a "fused" output item. In streaming environments, this is a continuous process, producing an output stream of fused items. As mentioned previously, such transformations can result in the expansion, contraction, or *status quo* in the data flow rate after the fusion. Note that a filter function, taking a single input stream and producing a single output stream, is a special case of such a transformation. We assume that fusion outputs can be shared by multiple consumers, allowing "fan-out" from a fusion point, but we disallow a fusion point with two or more distinct output streams. Fusion points with distinct output streams can be easily modeled as two separate fusion points with the same inputs, each producing a single output. Note that the input of a fusion point may be the output of another fusion point, creating fusion pipelines or trees. Fusion computations that implement control loops with feedback create cyclic fusion graphs.

The Fusion Channels package aims to simplify the application of programmer-supplied transformations to correlated sets of input items from sequenced input streams, producing a (possibly shared) output stream of "fused items." It does this by providing a high-level API for creating, modifying, and manipulating fusion points that subsumes certain recurring concerns (failure, latency, buffer management, prefetching, mobility, sharing, concurrency, etc.) common to fusion environments such as sensor networks. Only a subset of the capabilities in the Fusion Channels package are currently used by DFuse.

The fusion API provides capabilities that fall within the following general categories:

Structure management: This category of capabilities primarily handles "plumbing" issues. The fundamental abstraction in DFuse that encapsulates the fusion function is called a *fusion channel*. A fusion channel is a named, global entity that abstracts a set of inputs and encapsulates a programmer-supplied fusion function. Inputs to a fusion channel may come from the node that hosts the channel or from a remote node. Item fusion is automatic and is performed according to a programmer-specified policy either on request (demand-driven, lazy, pull model) or when input data is available (data-driven, eager, push model). Items are fused and accessed by timestamp (usually the capture time of the incoming data items). An application can request an item with a particular timestamp or by supplying some wildcard specifiers supported by the API (such as *earliest item*, *latest item*). Requests can be blocking or non-blocking. To accommodate failure and late arriving data, requests can include a minimum number of inputs required and a timeout interval. Fusion channels have a fixed capacity specified at creation time. Finally, inputs to a fusion channel can themselves be fusion channels, creating fusion networks or pipelines.