USC/ISI Technical Report 2002-556

Diffusion Filters as a Flexible Architecture for Event Notification in Wireless Sensor Networks

John Heidemann, Fabio Silva, Yan Yu, Deborah Estrin, Padmaparna Haldar

Abstract—**Wireless sensor networks represent an increasingly important example of distributed event systems. Unlike Internet-based distributed event systems, sensor networks are very bandwidth constrained and use sensor nodes that are often dedicated to the network and controlled by a single organization. Bandwidth constraints require, and administrative homogeneity allows, sensor networks to employ in-network processing, where application-specific code is used in the network to optimize data movement. The contribution of this paper is to describe the *diffusion filter architecture*, a software structure for a distributed event system that allows user-supplied software to interact with event routing. Sensor network nodes will span a wide range of capabilities, from tiny single-address space embedded processors to to desktop-class 32-bit computers. A second contribution of our architecture that it scales from 16- to 32-bit computers with OS support for single or multiple address spaces. We describe what software approaches facilitate this flexibility and quantify the performance differences.**

Keywords: Functionality and APIs of event services and publish/subscribe systems; Design, architecture, and engineering of event-based applications; Algorithms for distributed event processing (e.g. filtering, routing, composition, ordering)

I. Introduction

Wireless sensor networks represent an increasingly important example of distributed event systems. Sensor networks provide a good solution to the problems posed by applications such as environment monitoring and tracking because they fundamentally change the sensing problem both at individual sensors and collaboratively. By using a flock of small, inexpensive sensors with wireless communication, individual sensors can be positioned to be physically close to the objects being sensed, simplifying the signal processing problem. By communicating, groups of sensors can collaborate to reduce noise and false detections.

From the point of view of systems design, wireless sensor networks are important because they have very different constraints than Internet-based distributed event systems. Internet-based systems are typically constructed using end-to-end mechanisms (either organized as peers, with a central server, or with some hybrid in between), are connected by high-bandwidth networks (possibly 30–50kb/s at the edge, but certainly more than 1Mb/s in the network "core"), and have relatively low latencies (less than 1 second between any nodes). In sensor networks, by comparison, the emphasis is on moving processing in-the-network (as opposed to end-to-end) to compensate for very low bandwidths (10–20kb/s maximum radios) and potentially long delays (100ms or more due to energy-conserving radio protocols such as TDMA [19] or S-MAC [28]). Moreover, Internet systems focus on adding an event notification system to an existing, very large network (the Internet) spanning many administrative domains, where end-nodes are already being used for many applications and possess relatively immutable operating systems. Sensor networks instead are relatively smaller networks of nodes (10s to 1000s) deployed by a single administrative entity, dedicated to a single or a few related applications, with substantial control over system software.

The difference in constraints of sensor networks compared to the Internet focus a very different software architecture. Many successful Internet systems can nearly ignore the network topology (for example, Chord [24] and FreeNet [9]), and find that a strictly end-to-end architecture is important for rapid deployment in the heterogeneous Internet. By contrast, the use of in-network processing is critical to sensor networks to reduce communications costs [19], [17], [13], since every packet sent brings that node closer to death.

Although the constraints of sensor networks force an architecture different from that used for event-notification services in traditional networks, some of the design lessons learned may be applicable there. For example, as Internet-based event services become very large and distributed, in-network processing can aid scalability.

We have previously described the routing [17] and naming [13] approaches used in directed diffusion. The contribution of this paper is to describe the *diffusion filter architecture*, an approach to allow user-supplied software to influence how data is moved through the network. Sensor network nodes will span a wide range of capabilities, from 8-bit computers in "smart dust" [27] to desktop-class 32-bit computers. A second contribution of our architecture that it scales from 16- to 32-bit computers with OS support for single or multiple address spaces. We describe what software approaches facilitate this flexibility and quantify the performance differences.

II. The Diffusion Filter Architecture

Directed diffusion is used to disseminate information in the distributed system [17]. *Filters* are software modules that process data as it moves through the network. Matching rules control which filters are triggered and how data sources and sinks are related. We have previously described matching rules and filters [13]; in this paper we focus on how filters can interact and how they can be configured for different platforms. For context, we briefly summarize prior work on diffusion, filters, and the matching rules here.

This work was supported by DARPA under grant DABT63-99-1-0011 as part of the SCAADS project, NSF grant ANI-9979457 as part of the SCOWR project, and was also made possible in part due to support from Cisco Systems.

John Heidemann, Fabio Silva, and Padmaparna Haladr are with USC/Information Sciences Institute, 4676 Admirality Way, Marina del Rey, CA, USA. Deborah Estrin and Yan Yu are with USC/ISI and also the Computer Science Department, University of California, Los Angeles, USA. E-mail: {johnh,fabio,yanyu,estrin,haldar}@isi.edu.

A. Directed Diffusion

Directed diffusion is a data communication mechanism for sensor networks [17]. Data sources and sinks use attributes to identify what information they provide or are interested in. The goal of directed diffusion is to establish efficient *n*-way communication between one or more sources and sinks. We illustrate directed diffusion's data-centric approach with a brief example where a user tracks animals.

A user's application (the data *sink*) begins communication by *subscribing* to information, specified by a combination of generic and application-specific attributes. These attributes identify the desired data, specifying for example the sensor types and region of interest. This subscription causes an *interest* with these attributes to be propagated through the network. As the interest travels through the network, each node establishes a *gradient*, state that represents where data should flow.

When the interest reaches an appropriate region, zero, one or more matching sensors there are activated, becoming *sources*. These sensors then generate *data* messages that flow back toward the source. Data messages are occasionally marked *exploratory*; these messages trigger *reinforcement messages* that select a low-latency path through the network. Non-exploratory messages travel only on these reinforced paths.

As data flows through the network it may be cached at intermediate nodes. Cached data is used for several purposes at different levels of diffusion. The core diffusion mechanism uses the cache to suppress duplicate messages and prevent loops, and it can be used to preferentially forward interests. Cached data is also used for application-specific, in-network processing. For example, data from detections of a single object by different sensors may be merged to a single response based on sensor-specific criteria, or data may be replayed from the cache if it was unsuccessfully transfered downstream.

B. Filters and Matching Rules

If directed diffusion is the underlying algorithm that allows data to move from node to node, *filters*, *attributes* and *matching rules* are the mechanisms that make it possible [13].

Filters are application-provided software modules that are allow applications to influence diffusion and data processing. Uses of filters include in-network aggregation, collaborative signal processing, caching, and similar tasks that benefit from control over data movement, as well as debugging and monitoring. We describe the specific filters we have implemented or designed in Section IV.

Data is specified as *attributes*. Each message consists of a list of key-value-operation attribute tuples. The *key* identifies the type of the attribute (latitude, sensor type, target, etc.), the value is its quantity (34.33, seismic, "four-legged animals", etc.). The *operation* specifies either a conditional (EQ, NE, LE, corresponding to equality, inequality, less than, etc.), a *formal* value, or IS, that specifies constant data, an *actual* value.

A message entering a node triggers a filter if the the attributes specified by the filter *match* the attributes in the message. Matching rules have been previously specified [13], [10]; briefly, each formal in one set of attributes must match some actual in the other set of attributes. For example, a user's request to search for certain animals might be the attributes (type EQ four-legged-animal-search, interval IS 20ms, duration IS 10 seconds, x GE −100, x LE 200, y GE 100, y LE 400), this would match a sensor with data (type IS four-legged-animal-search, instance IS elephant, x IS 125, y IS 220, intensity IS 0.6, confidence IS 0.85, timestamp IS 1:20, class IS data).

Apps and filter APIs:

```
handle NR::subscribe(NRAttrVec *subscribeAttrs,
                     const NR::Callback * cb);
int NR::unsubscribe(handle subscription_handle);
handle NR::publish(NRAttrVec *publishAttrs);
int NR::unpublish(handle publication_handle);
int NR::send(handle publication_handle,
             NRAttrVec *sendAttrs);
```

Filter-specific APIs:

```
handle addFilter(NRAttrVec *filterAttrs,
          int16_t priority, FilterCallback *cb);
int NR::removeFilter(handle filter_handle);
void sendMessage(Message *msg, handle h,
                 int16_t priority = 0);
```

Fig. 1. Basic diffusion APIs.

III. Diffusion APIs

We have developed a very simple API for applications and filters (see [10] for a complete specification and example source code, and [13] for a description of an earlier version of these APIs). The APIs define a publish/subscribe approach to originating data, plus mechanisms for filters. Figure 1 shows the C++ APIs.

To receive data, nodes *subscribe* to particular set of attributes. A subscription results in interests being sent through the network and sets up gradients. A callback function is then invoked whenever relevant data arrives at the node.

Applications that generate information *publish* that fact, and then *send* specific data. The attributes specified in the publish call must match those of prior subscriptions. If there are no active subscriptions, published data does not leave the node.

There are also operations to add and remove filters, and for filters to forward messages between each other. Although logically messages pass from filter to filter, in practice all messages pass through the diffusion core which shepherds messages from filter to filter as specified by filter *priorities*. SendMessage uses filter priorities to indicate how filters are ordered when they are configured, and how messages should pass through the filter stack. Priorities are simple integers that define a total ordering of filters (duplicate priorities are not allowed). Because filters can arbitrarily manipulate messages (changing the attributes or suppressing or sending additional messages), the "set" of relevant filters may change as the message moves through the system. We describe how priorities are handled in Section V-A in light of our experiences with multi-filter configurations.

IV. Currently Designed and Implemented Filters

Filters are an important part of the diffusion architecture because they provide a modular way to allow application-specific code to influence event routing. The important design question in filter modularity is to select an API that is rich enough to let

filters accomplish what they need to, but one that hides enough data that filters do not interfere with each other. This section lists the modules we have written or designed and we then consider the filter API we settled upon. In Section V we then evaluate these choices.

Figure 2 shows the set of filters that we have implemented and designed. The diffusion core interacts with all filters (rectangles), applications (circles at the top right), and radio hardware (the lozenge at the bottom). Solid and dashed rectangles represent existing and planned filters, respectively. The core is responsible for dispatching all messages as they pass through the system and for suppressing duplicate messages.

A. Filters

Basic diffusion is implemented in the *gradient* filter. This filter maintains gradients representing the state of any existing flows to all neighbors and is responsible for periodically sending out reinforcement messages and interests.

GEAR is a pair of filters that can optionally surround diffusion to implement Geographic and Energy-Aware Routing [29]. Lacking prior information (such as geographic information or prior saved state), basic diffusion floods interests to all nodes in the network. GEAR overrides this behavior to forward messages with geographic assistance (interests are sent basically toward their geographic destination, but around any holes in the topology). GEAR consists of two filters, a *pre-processing* filter that sits above the gradient module to handle GEAR-specific beacon messages and to remove transient geographic information on arrival, and a *geographic routing* filter that acts after the gradient model to forward interests in a good direction. The two filters that make up GEAR share information (such as the list of neighbors); we describe this unusual structure and how it illustrates the flexibility of our filter architecture in Section V-C.

Reliable big blob is a module that allows reliable transfer of large (multi-packet), uninterpreted data across unreliable links. It caches data locally to support loss recovery, similar to approaches taken in reliable multicast [12] and SNOOP TCP [2]. We are currently implementing this filter.

The *information-driven tracking filter* is an example of how application-specific information can assist routing proposed by researchers at Xerox PARC [30]. An important application of sensor networks is object tracking—multiple sensors may collaborate to identify one or more vehicles, estimating their position and velocity. Which sensors collaborate in this case is dependent on the direction of vehicle movement. They have proposed using current vehicle estimates (or "belief state") to involve the relevant sensors in this collaboration while allowing other sensors to remain inactive (conserving network bandwidth and battery power). While GEAR uses generic (geographic) information to reduce unnecessary communication, the information-driven tracking filter uses application-specific information to further reduce communication. As other applications are explored we expect to develop other application-specific filters similar to the information-driving tracking filter.

One use of filters is logging information for debugging. We have implemented a *logging* filter for this purpose, and we are considering implementing an *ns-logging* filter for simulator-specific logging. These filters are shown to the left of diffusion stack because they can be placed between any two modules.

Although this architecture was built to explore diffusion-style routing, for debugging purposes we also developed support for source routing. Source routing is provided as two filters: the *source tagging* filter functions similar to the logging filters in that it can be configured anywhere in the diffusion stack. This filter adds a record of each node that the message passes through, much like the `traceroute` command used on the Internet. The *source routing* filter provides the opposite function. It takes a message that includes an attribute listing the path of nodes the message should take through the network and dispatches it along that path. A design principle of directed diffusion is local operation—nodes should not need to know information about neighbors multiple hops away. While source routing is directly opposite to this goal, it can be provided within our software framework, and is still sometimes a useful debugging tool.

B. Communications Modules and Applications

In addition to interacting with filters, the diffusion core interacts with user applications and communications devices such as radios.

We currently support three different radios: the Radiometrix Radio Packet Controller, an off-the-shelf packet-based radio at 418MHz, providing about 13kb/s throughput; MoteNICs, a UC Berkeley Mote [15] with an RF Monolithics radio running software designed at UCLA to operate as a network interface card; and the Sensoria WINSng 2.0 radios, a TDMA-based radio providing about 20kb/s throughput. For desktop development, TCP- or UDP-based links can substitute for radios.

The diffusion core also dispatches messages to applications. Both applications and filters identify the messages they wish to receive with attributes and matching; the primary difference between them is that applications can only send messages to the "top" of the diffusion stack while filters can send messages from filter to filter. We describe this process next.

V. Experiences with Filters

The major contribution of this paper is a better understanding of how filters interact and what they are capable of. Although we have previously defined the Filter APIs, recent experience with multiple filters provides a more complete picture about filter operations. Below we describe our conclusions about how filter ordering should operate, how diffusion can be configured to run in single and multiple address spaces (for PC-class or very small computers), how much filters can influence the basic diffusion algorithms.

A. Filter Ordering

Priorities, defined at filter configuration, give a total ordering of all filters in a system. While message attributes select which filters can process a message, priorities specify the *order* in which those filters act.

Priorities are needed because the attributes of an incoming message may match multiple filters. In this case, filter priorities indicate which filter is invoked first. That filter will call sendMessage to forward the message on. By default, if that filter doesn't specify a priority in sendMessage, the message goes

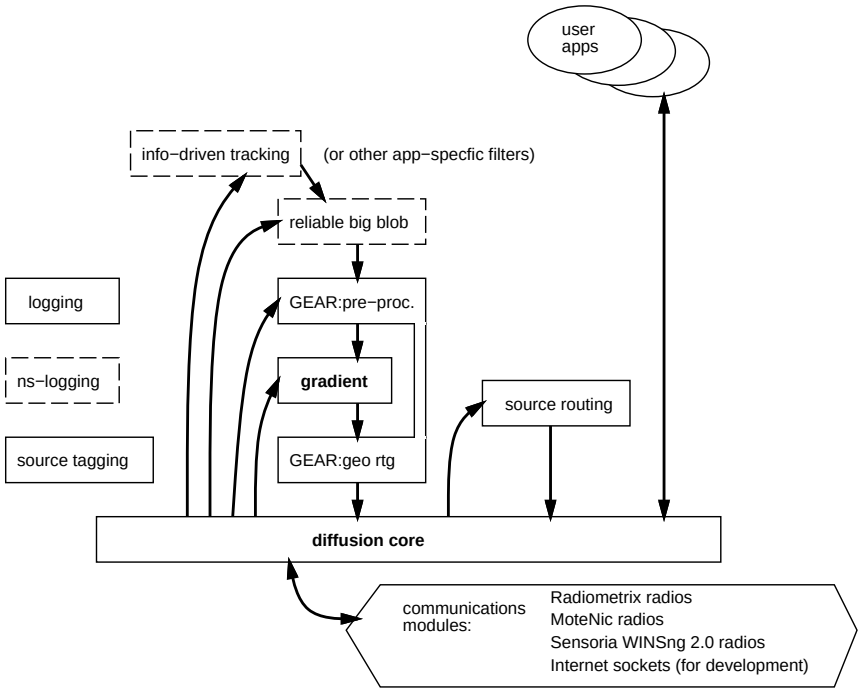

Fig. 2. Current and planned filters in diffusion and how they interact.

to the filter with the next lower priority. For example, in Figure 2, an interest message that includes geographic information would be passed to the GEAR pre-processor with highest priority, that would forward it to the gradient filter and on to the GEAR geographic routing filter.

Although this simple case handles a single message passing through a series of filters, filters can do any kind of processing on receipt of a message, including responding with multiple messages or changing message attributes in a way that the set of relevant filters changes. For example, consider a message containing a segment of data that arrives at the the reliable big blob filter. Information in this message might indicate that a prior segment of data was lost, so in addition to forwarding the new message the filter may send a negative acknowledgment message asking for replay of the missing data.

The sendMessage API allows filters to override the order of message processing by changing the priority field or message attributes. Thus a knowledgeable filter can direct a message anywhere in the diffusion stack. Since the contents or priority can change any time a message leaves a filter, all messages are always sent to the diffusion core, not immediately to the next filter. Thus the arrows connecting filters in Figure 2 represent the typical and logical message path, not the exact or literal path. The diffusion core recomputes the next filter each time it receives the message. (An obvious optimization that we have not yet implemented is to cache this computation for the common case where the filter list does not change.)

B. Filter Operation in Single and Multiple Address Spaces

We expect diffusion to operate with all filters and applications in a single address space where necessary, but we would also like it to operate in multiple address spaces where possible. This section describes how we are able to do that and how performance compares with each option.

B.1 The need for both options

Filters play an important role in our architecture, allowing application-specific code to be deployed throughout the network. They are used to minimize communication costs with techniques such as data aggregation, and to improve sensor effectiveness with collaborative signal processing. An important design question when introducing user-provided code into a system is how to isolate that code from the rest of the system. Operating systems provide the abstraction of a *process* to compartmentalize software, allowing individual components to fail independently (in the case of bugs) and preventing them from accidentally or maliciously interfering with each other.

We expect diffusion to run over a wide range of sensor hardware, from very small hardware where there may be no support for multiple address spaces, or it where it is too expensive, to larger platforms where such protection is available. Sensoria WINSng 1.0, one of our early platforms ran Windows CE with all components in a single process[1]. More recent platforms including WINSng 2.0 nodes and our PC/104 nodes were able to support multiple address spaces, greatly simplifying development and debugging.

B.2 Filter structure to support both options

Filters move relatively smoothly between single or multiple address spaces due three software design choices. First, we structured the software to be completely event driven, rather than multi-threaded. Flow-of-control is centralized in an event

[1]Windows CE supports multiple processes, but hardware-specific libraries forced us into a monolithic architecture.

loop, I/O and timers are provided with callbacks. This approach allows us to avoid explicit contention for data structures and isolates the diffusion algorithms from direct interaction with the operating system.

Second, we implemented a small IPC system to move data between processes when filters run in multi-process model. Because diffusion already needs to marshal data to send it between different nodes, there was little additional code required to use this mechanism between different processes on the same machine. As an optimization, marshaling should not be used when all filters run in a single address space. We are in the process of implementing this optimization.

Finally, we had to avoid unqualified global state, since global variables for different filters could conflict. Filters are implemented in C++, so a simple solution is to keep all filter state as instance variables of the class.

An unexpected benefit of our design to support single- or multiple-address space operation is that it was quite easy to port diffusion to run inside the ns network simulator [6], that is structured as a single-process, event-driven simulator.

Although our implementation was able to support both single and multiple address spaces and 16- to 32-bit computers, it is not currently possible to scale this approach directly down to 8-bit embedded processors with working memories measured in 100s of bytes. Micro-diffusion subsets the diffusion filter architecture, providing only a few active gradients, a very small packet cache, and hard-coded filters [13]. This class of compile-time subsetting is necessary to scale to these very small platforms. Although a complete re-implementation as a TinyOS component [15], it shares many of the design principles of full diffusion.

B.3 Relative Performance

Since we were able to provide both single- and multiple-address space option, we are interested in comparing the performance of each configuration. To evaluate processing performance, we sent message through diffusion stacks where they match from 1 to 5 filters. For simplicity we used multiple copies of the logging filter (configured at different priorities), since it records a simple message and passes the message unchanged to the next filter. We measured the time it takes for a packet to start from a local application, pass through all the log filters, and then pass through a lowest filter that records the time the message exits the system just before going out the radio. (We have previously described basic matching performance in a single address space [13].)

In this experiment, we used a 160-byte long, 8-element data message, whose attributes are shown in Figure 3. Note that in each message, the time attribute contained the actual time the message was sent by the local application. This experiment was done on our Sensoria WINSng 2.0 sensor node, with a 166Mhz Hitachi SH-4-class CPU, 32MB of Flash storage and 64MB of RAM. Because message processing is quite fast, even on this relatively slow machine, we sent 2000 messages and plot the mean transit time of each message. (This approach may slightly overestimate the speed of the system due to favorable memory cache effects.) We also show 95% confidence intervals, even though they are less than 5% of the mean for all points in the graph.

```
class IS data
scope IS global
latitude IS 60.0
longitude IS 130.0
target IS “4-leg”
det_id IS 4
time IS TIME
task EQ “detectAnimal”
```

Fig. 3. Attributes used for filter performance experiments.

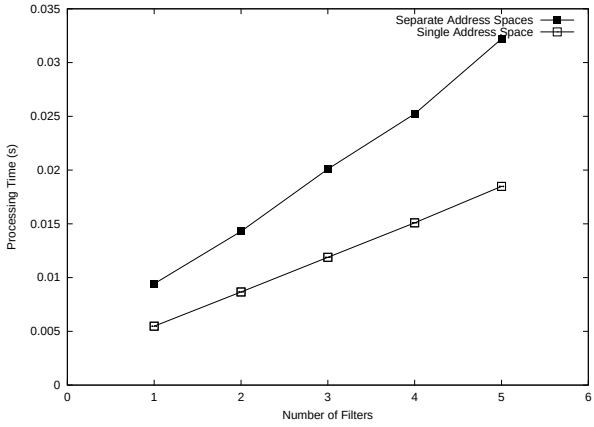

Fig. 4. Processing performance as the number of filters grow.

Our expectation is that the processing cost of filters is linear with the number of filters matching the incoming message. This is confirmed in Figure 4 that shows the processing cost of filters as the number of filters matching the message increases from 1 to 5. The lower line shows the case where all pieces of software are compiled in a single application and run in the same address space. The upper line shows the case where each piece of the code (the main diffusion module, the local application sending the messages, the filter collecting the messages at the end and each log filter) each run in a separate address space.

We also observe the performance benefits of operating in a single address space: transit time for the system with multiple address spaces is about twice that of the monolithic configuration. On some very small hardware platforms without memory protection, this ability to run in a single address space is a necessity.

Although twice the performance is an improvement, we expected single-address-space operation to be considerably faster. Part of the reason gains are currently modest is that the current implementation does full data marshaling and a fair amount of copying even when all processes run in the same address space. While this implementation choice speeded development, it greatly reduces single-address-space performance. We are in the process of removing this unnecessary overhead.

We suggest that it is also important to have the option of the opposite configuration. Sensor networks will depend on application-specific code for efficient communication, and as embedded systems they must remain highly available in spite of software failures. For hardware platforms powerful enough to provide multiple address spaces, the ability to isolate user-supplied code in a separate address space provides an important

level of robustness since individual filters can fail but the overall system can keep functioning.

C. Other Comments About Filters

We have designed the diffusion filter architecture to support a very high-level of user configurability. An important question is if we have accomplished this goal. Further experience is required here, but so far our experiences have been promising.

GEAR provides a good example as a filter that fundamentally changes how the gradient module processes messages. GEAR requires two filters, a pre-processing filters above the gradient to handle beacon messages, and a geographic-routing module below diffusion to redirect outgoing interests. Early in the diffusion architecture design we were concerned that GEAR would require detailed access to internal gradient data structures and so the two necessarily be closely intertwined. Fortunately, we were able to avoid that loss of modularity at with only slight duplication of information: both GEAR and the gradient module keep lists of neighboring nodes.

Another challenge of GEAR is that the two haves of GEAR need to share information and data structures. We chose to place both filters in the same address space to allow shared data structures. Since each filter specifies a different callback function, this structure is easily supported, even if the gradient module runs in a separate address space.

D. Future Directions

Although we are happy with our current architecture, some questions remain unresolved. An important open question is how to mitigate duplication of data structures and caches in different filters. Several filters benefit from neighbor information that currently is maintained separately. As we develop additional application-specific filters we expect to see an increasing need to cache message contents in different filters. Cache sizes will be maximized if caches in different filters can be shared.

A more general question is more sophisticated control interfaces will be required between modules. For example, if neighbor lists were provided directly, a new API might cause callbacks when the set of neighbors changed. These APIs can be approximated with internal messages and attributes, but it is not clear how effective that will be in the long term.

Another option to explore is run-time deployment of new filters. We currently pre-configure filters when nodes are started, or manually reconfigure them between runs. Although we have not yet experimented with it, run-time deployment of new filters is both possible and fairly easy when filters run in multiple address spaces. On-the-fly deployment of filters would require reliable delivery of the code (possibly with our "reliable big blob" module) and security provisions such as signed code or safe execution environments such as have been explored in active networking [25].

VI. Related Work

The work described in this paper builds on a very broad base of related work, from existing distributed event systems to previous approaches to structure network and operating systems, and on other sensor network systems. Due to space limitations we summarize only the most closely related work here.

A. Distributed Event Systems

The publish/subscribe paradigm has long been studied as the basis of a distributed event system. Linda proposed structuring distributed programs using several CPUs around an attribute-indexed common memory called a tuple space [7]. For the S/Net implementation this was the basic communication mechanism, but proposed implementations assume uniform and rapid communications between all processors. Later systems such as ISIS [4] and the Information Bus [18] provide a "publish and subscribe" approach where information providers publish information and clients subscribe to attribute-specified subsets of that information. These systems are designed to be robust to failure, but again assume reasonably fast, plentiful, and expensive communications between nodes.

More recently, systems such as Sienna [8] have focused on wide-area systems where data bandwidth is of concern; independently adopting an attribute based approach very similar to ours. Content routing systems such as that of Snoeren et al. adopt an XML-based filtering approach [23]. Like our system, these systems allow the user to filter data in the network; a major difference however is their focus on Internet applications with relatively high bandwidth while we target sensor networks with bandwidths often less than 20kb/s.

A second trust of research research has focused on peer object location system where events may be very simply identified (perhaps by a hash value) and are distributed across the network. Examples include FreeNet [9] and Chord [24]. One application of these systems is a distributed, peer-to-peer storage system. Unlike these systems, we focus on communication of transient, not persistent data, and in sensor networks locality is quite important while these systems are much less concerned with network locality.

Another class of distributed event systems are resource discovery systems. Recent examples include as Jini [26], Ninja [11], and INS [1]. These systems typically focus on support for frequent device arrival and departure and support for user interaction and operating system configuration. Unlike our system, they are often less limited by network capacity.

B. Software Structure for Communications and Operating Systems

How to structure system software such as operating systems and communications has been a topic of study for the last thirty years. Relevant related work from the communications domain includes Streams [20] and the x-kernel [16], as well as from other fields such as databases [3] and file systems [21], [14], each of which faced similar problems in module configuration. Although many of the issues are similar, our approach to routing messages through filters based on possibly changing message contents is much more dynamic than most of these systems.

Microkernels face the similar issues to running in single or multiple address spaces as we do. Systems such as Mach [5] argued for the benefits of running modules in separate address spaces and suggested many ways to optimize IPC. The Chorus System [22] provided some flexibility as to where individual components could run. Unlike our work they were not forced to support single address space operation to run on very low-end

hardware.

VII. Conclusions

We have described our experiences using the diffusion filter architecture. Our target application domain is sensor networks, where limitations of network bandwidth encourage wide use of application-specific code. Our growing experience with the system suggests that this architecture is flexible enough to handle a range of filters and hardware platforms.

Acknowledgments

Ramesh Govindan also contributed to the design of the diffusion APIs and code structure; the authors thank him for his contributions. We would also like to thank Fred Stann who is currently implementing the Reliable Big Blob module.

References

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proceedings of the 17th Symposium on Operating Systems Principles*, pages 186–201, Kiawah Island, SC, USA, December 1999. ACM.

[2] Hari Balakrishnan, Srinivasan Seshan, Elan Amir, and Randy H. Katz. Improving TCP/IP performance over wireless networks. In *Proceedings of the First ACM Conference on Mobile Computing and Networking*, pages 2–11, Berkeley, CA, USA, November 1995. ACM.

[3] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. *IEEE Transactions on Software Engineering*, 14(11):1711–1730, November 1988.

[4] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, December 1993.

[5] David L. Black, David B. Golub, Daniel P. Julin, Richard F. Rashid, Richard P. Draves, Randall W. Dean, Alessandro Forin, Joseph Barrera, Hideyuki Tokuda, Gerald Malan, and David Bohman. Microkernel operating system architecture and Mach. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 11–30, April 1992.

[6] Lee Breslau, Deborah Estrin, Kevin Fall, Sally Floyd, John Heidemann, Ahmed Helmy, Polly Huang, Steven McCanne, Kannan Varadhan, Ya Xu, and Haobo Yu. Advances in network simulation. *IEEE Computer*, 33(5):59–67, May 2000. (An expanded version is available as USC CSD TR 99-702b.).

[7] Nicholas Carriero and David Gelernter. The S/Net's Linda kernel. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 110–129. ACM, December 1985.

[8] Antonio Carzaniga, David S. Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, August 2001.

[9] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage retrieval system. In *Proceedings of the ICSI Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, USA, July 2000.

[10] Dan Coffin, Dan Van Hook, Ramesh Govindan, John Heidemann, and Fabio Silva. *Network Routing Application Programmer's Interface (API) and Walk Through*. MIT/LL and USC/ISI, December 2000.

[11] Steven E. Czerwinski, Ben Y. Zhao, Todd D. Hodes, Anthony D. Joseph, and Randy H. Katz. An architecture for a secure service discovery service. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 24–35, Seattle, WA, USA, August 1999. ACM.

[12] S. Floyd, V. Jacobson, C. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the ACM SIGCOMM Conference*, pages 342–356, Cambridge, Massachusetts, August 1995. ACM.

[13] John Heidemann, Fabio Silva, Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, and Deepak Ganesan. Building efficient wireless sensor networks with low-level naming. In *Proceedings of the Symposium on Operating Systems Principles*, pages 146–159, Chateau Lake Louise, Banff, Alberta, Canada, October 2001. ACM.

[14] John S. Heidemann and Gerald J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1):58–89, 1994. Preliminary version available as UCLA technical report CSD-930019.

[15] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for network sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, Cambridge, MA, USA, November 2000. ACM.

[16] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.

[17] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, USA, August 2000. ACM.

[18] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The Information Bus—an architecture for extensible distributed systems. In *Proceedings of the 14th Symposium on Operating Systems Principles*, pages 58–68, Asheville, North Carolina, USC, December 1993. ACM.

[19] Gregory J. Pottie and William J. Kaiser. Embedding the internet: wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[20] Dennis M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, October 1984.

[21] David S. H. Rosenthal. Evolving the vnode interface. In *USENIX Conference Proceedings*, pages 107–118. USENIX, June 1990.

[22] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Chorus distributed operating systems. *Computing Systems*, 1(4):305–370, Fall 1988.

[23] Alex C. Snoeren, Kenneth Conley, and David K. Gifford. Mesh-based content routing using XML. In *Proceedings of the Symposium on Operating Systems Principles*, pages 160–173, Chateau Lake Louise, Alberta, Canada, October 2001. ACM.

[24] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference*, Stockholm, Sweden, September 2000. ACM.

[25] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *ACM Computer Communication Review*, 26(2):5–18, April 1996.

[26] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(10):76–82, October 1999.

[27] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S.J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *IEEE Computer*, 34(1):44–51, January 2001.

[28] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, page to appear, New York, NY, USA, 2002. USC/Information Sciences Institute, IEEE.

[29] Yan Yu, Ramesh Govindan, and Deborah Estrin. Geographical and energy aware routing: A recursive data dissemination protocol for wireless sensor networks. Technical Report TR-01-0023, University of California, Los Angeles, Computer Science Department, 2001.

[30] Feng Zhao, Jaewon Shin, and James Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, page to appear, March 2002.