Building Autonomic Routing Services for Sensor Networks

Yu He, Cauligi S. Raghavendra

Information Sciences Institute
University of Southern California
4676 Admiralty Way, Suite 1001
Marina del Rey, CA 90292
U.S.A.

Abstract

Current routing services for sensor networks are application-specific. They are preconfigured into nodes, and thus have difficulty in adapting to different applications and network conditions. This paper describes a framework to build autonomic routing services for sensor networks. This framework is based on a well-defined parameter space of routing services, and includes a programmable service interface and a deployment module. The programmable interface can implement any routing service that is configured by the deployment module in an automatic and energy-efficient way. In addition, this framework facilitates a monitoring mechanism to adapt routing service parameters. This paper presents implementation details of this autonomic framework and shows simulation results.

1 Introduction

Technological advances have led to the emergence of small, low-cost, and low-power devices that integrate micro-sensing with on-board processing and wireless communication capabilities ([1], [2], and [3]). These sensors can be deployed in a large scale and deeply embedded within the physical environment. Each sensor node can operate autonomously with no central point of control and can make decisions based on the information it currently has. Networks composed of these sensor nodes can support many new applications

Email addresses: `yuhe@usc.edu` (Yu He), `raghu@usc.edu` (Cauligi S. Raghavendra).

such as physiological monitoring, environmental monitoring (air, water, soil, chemistry), precision agriculture, transportation, factory instrumentation and inventory tracking, condition based maintenance, smart spaces, and military surveillance ([4], [5], [6], [7], and [8]).

The goal of a sensor network is to collect, process, and forward sensed data to other sensor nodes and/or base stations. Therefore, a routing service is essential to sensor network applications. Several routing services have been proposed for sensor networks. They include Greedy Perimeter Stateless Routing (GPSR) [9], Geographical Energy Aware Routing (GEAR) [10], Trajectory Based Routing (TBF) ([11] and [12]), Directed-diffusion [13], Two-Tier Data-Dissemination (TTDD) [14], Content-Based Multicast (CBM) [15], Rumor-routing [16] and others. These routing services have distinct properties than those traditional routing such as OSPF [17] and RIP [18]. First, they try to meet the resource-limited requirements of sensor networks [1]. Second, data may be forwarded in sensor networks without explicitly addressing individual nodes. For example, GPSR takes locations instead of IP addresses as routing destinations. TBF uses a trajectory instead of a node-path as its routing unit. Further, many routing services, such as Directed-diffusion, TTDD, and CBM, forward packets according to packet contents. In addition, routing services in sensor networks can manipulate the forwarded data (such as aggregation and fusion[19]).

Designing efficient routing services for sensor networks often needs to consider the properties of specific applications. It is thus unlikely to have a fixed routing service in sensor networks that can effectively meet the requirements of all applications. For example, Directed-diffusion performs well when collecting data from static sources. When the sources become mobile, TTDD is a better choice. Furthermore, if sinks need to aggressively collect urgent data, CBM can be a more suitable solution than TTDD. Note that the adaptability of some existing routing services that react on certain network state (such as link loss) is not enough to accommodate the above changes of the properties of applications. There are times when changes of network conditions may even make it necessary to reform the routing service to a "new" one. For example, GPSR is an operational routing service when there is a scalable location service, but it cannot be used when the location service becomes not available. In this case, if the sensor nodes are densely deployed, Rumor-routing can be a feasible routing choice.

Currently, it is very difficult, if not impossible, to change a routing service in a sensor network because the service code and parameters are statically pre-configured into each node, which is often unattended.

Our goal is to have a routing service that can adapt to different applications and network conditions, whose rich parameter space allows enough room for adaptability. Individual nodes can autonomously change service parameters according to available information. In case human interception is necessary, such changes should be made as automatically as possible. The adaptations should be done with low energy consumption since energy is a scarce resource in sensor networks. We call such a routing service *autonomic* [20] in the sense that application changes are automatically accommodated and management complexity is significantly reduced.

The autonomic routing service for sensor networks is built by following a framework we describe in this paper. This framework is based on a well-defined parameter space for routing services of sensor networks that explores more service behavior than traditional plain parameters. Different routing services are covered by this unified parameter space. Thus changing parameters of a routing service can lead to a "new" service. The routing service architecture provided by the autonomic framework includes a service interface, and a deployment module. The service interface indicates implementation specifications of any routing service that is configured by the deployment module in an automatic and energy-efficient way. In addition, the architecture facilitates parameter-tuning by a monitoring component that associates service performance and behavior with the parameter space.

This paper is organized as follows. Section 2 defines the parameter space of an autonomic routing service for sensor networks. Section 3 discusses the service architecture. In Section 4, we presents the approaches we take and decisions we make to implement the autonomic framework. Then Section 5 shows preliminary simulation results for our implementation. Section 6 describes related work. Finally, concluding remarks and future work are given in Section 7.

2 Parameter Space for Autonomic Routing Services in Sensor Networks

Each existing routing service for sensor networks has its own parameters. Table 1 lists sample parameters for existing routing services. In Table 1, some of the parameters are similar (e.g., the beacon interval used when obtaining neighbor information), others are distinct. Changing these parameters may tune the performance of routing services.

The problem with existing sets of parameters is their limitation in characterizing the corresponding services. The characteristics of each existing routing service are mainly written into its implementation code. Having a set of parameters based on which the main behavior of a routing service can be decided is a key step to automate the service since these parameters enable the service to describe itself and the service can adapt its behavior significantly by simply

	Sample Parameters
GPSR	Beaconing interval, Desynchronizing terms, Perimeter-mode flag, Planarization flag
GEAR	Beacon request delay, Beacon reply delay, Energy threshold, Neighbor number threshold, Path length threshold
TBF	Beacon delay, Neighborhood range, Sampling-space distance
Directed-diffusion	Interest delay, Gradient delay, Neighbor timeout, Exploratory data delay, Reinforcement sending delay, Data forwarding delay, Subscription delay
TTDD	Beacon delay, Soft-state refreshing interval, Cell size
CBM	Push interval, Push range, Pull interval, Pull range
Rumor-routing	Agent generation probability, Agent TTL, Query TTL, Retransmission delay

Table 1
Sample Parameters of Existing Sensor Routing Services

changing these parameters.

We propose a concept called *parameter space* to cover all necessary parameters for an autonomic routing service. The parameter space expands the traditional meaning of a parameter set for a routing service. It includes three categories of parameters. The first one is called *unit parameters* that reflect a single value for some performance purpose. These parameters contain traditional ones (such as those in Table 1). The second category is *structural parameters* that consist of unit parameters. They can also be constructed recursively. The third category is called *programmable parameters* that reflect those with specific functions.

There are other unit parameters than those traditional ones included in the proposed parameter space. One example is the packet types used in a routing service. Although this part of parameters may not relate to the performance of an service, their changes do alter the behavior of the service.

The structural parameters provide a means to represent a logic collection of parameters. Generally the components of a structural parameter can be both unit ones and structural ones, but not programmable ones. Structural parameters have two functions for our proposed autonomic services. First, a

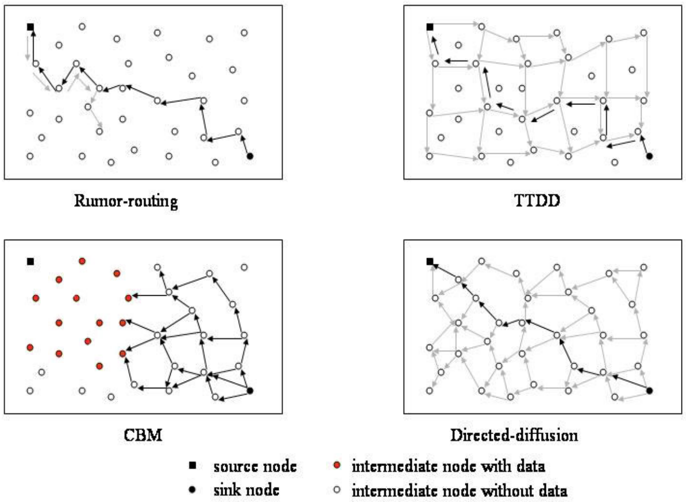

Fig. 1. Visiting Patterns of Control Packets for Four Routing Services

structural parameter gathers components together into a single entity that is easy to be deployed. Second, it represent somewhat complex characteristic of a routing service. One generic example is that the state of a routing service can be expressed via a structural parameter. One specific example is that a certain trajectory in TBF [11] can be characterized by a structural parameter.

The programmable parameters bring most freedom to some parts of a routing service. This category of parameter can include not only a whole packet handler, but also small programmable components. For example, instead of using a structural parameter to express certain trajectory in TBF, we can use programmable parameters to represent arbitrary trajectories. Another example is that the data fusion part of a routing service can be expressed as a programmable parameter. As for the relationship of programmable parameters to the other two, the programmable parameters often use the unit parameters and structural parameters.

One important parameter that we have observed is *visiting pattern* of packets. This parameter indicates what decision a routing service is supposed to make to forward a packet. Similar to the trajectory parameter in TBF, the visiting-pattern parameter can be simply a structural parameter, or a programmable component to make arbitrary decisions. This parameter has important implications for the communication overhead of a routing service, and tuning it is essential to the service performance. Fig. 1 shows four example visiting patterns. Table 2 lists data-forwarding patterns for existing routing service. As described in Fig. 1 and Table 2, different visiting-pattern parameters lead to significantly different service behavior.

There is a need for a concise language to convey the proposed parameter space. XML [21] is good at expressiveness, but is verbose, thus is not a suitable

	Data-forwarding function
GPSR	Closest or perimeter neighbor to destination
GEAR	Closest to destination(s) or multiple neighbor(s)
TBF	Neighbor(s) along trajectory
Directed-diffusion	Interested neighbor(s) with requested rate(s)
TTDD	Interested grid cross-point neighbor(s)
CBM	Interested neighbor(s)
Rumor-routing	Next hop in routing table

Table 2
Data-forwarding Functions in Sensor Routing Services

language. The following is a simplified version of our proposed language shown with a BNF-like syntax.

```
<ParameterSpace>  ::=   <UnitParameter> [<sp> <ParameterSpace>] |
                        <StructuralParameter> [<sp> <ParameterSpace>] |
                        <ProgrammableParameter> [<sp> <ParameterSpace>]
<UnitParameter>   ::=   <ParameterName> "=" <ParameterValue>
<StructuralParameter>  ::=<UnitParameter>                            |
                          <StructuralCharater> <ParameterName> "=" "("
                          <StructuralParameter> [<StructuralParameter>] ")"
<ProgrammableParameter>::=<ProgrammableCharater> <ClassName> "=" <Code>
<StructuralCharater>   ::= "$"
<ProgrammableCharater> ::= "@"
<ClassName>       ::=   <Fully-qualified C++ class name>
<ParameterValue>::=   <StringValue> | <Number>
<ParameterName> ::=   <StringValue without beginning with "$" or "@">
<StringValue>   ::=   <letter> {<letter> | <digit>}
<Code>          ::=   <0> | <1> | <0> [<Code>] | <1> [<Code>]
<Number>        ::=   <digit> {<digit>} {"."} {<digit>}
<sp>            ::=   <1 space char>
```

Using the above definition, we can easily express complex parameters. One example is a generic state entry for an autonomic routing service. Suppose the service maintain a state entry for each of its neighbor that includes the following fields:

- neighbor description (id, location, direction, distance, and energy reading)
- neighbor interest (type, rate, and duration)

- neighbor data availability (type, and duration)
- neighbor's latest data copy (data and timestamp)

Then the corresponding structural parameter can be expressed as follows:

```
$NeighborEntry=(
        $NeighborDescription=(
                NeighborId=<some number>
                NeighborLocation=<some number>
                NeighborDirection=<some number>
                NeighborDistance=<some number>
                NeighborEnergy=<some number>
        )
        $NeighborInterest=(
                DataType=<some string>
                DesiredRate=<some number>
                ExpectedDuration=<some number>
        )
        $NeighborDataAvailability=(
                DataType=<some string>
                ExpectedDuration=<some number>
        )
        $NeighborDataCopy=(
                DataContent=<some number>
                TimeStamp=<some number>
        )
)
```

As we mentioned before, the programmable parameters use the unit and structural parameters when carrying out their functions. Thus it is up to the programmable components of a routing service to assign actual meanings to the unit and structural parameters. Note that there are two choices about how to make up a parameter name. It can be either any string defined above or from some reserved names. For the first choice, any instance of an autonomic routing service can choose its own set of parameters. The explanations to the parameters are up to individual service instances. For any service instance, it is easy to extend existing parameters. For the second choice, the system reserves in advance specific names for the parameter space. All parameters of a service instance are chosen from the reserved names. A parameter name has the same meaning in all service instances. Since these reserved names can be easily encoded into concise formats, communication overhead for transferring these parameters is lower than the first way. But this choice is less flexible to extend the parameter name space. Currently the former choice is used in our implementation of the autonomic framework.

We consider the proposed parameter space an essential part of our autonomic routing framework for sensor networks. The parameter space contains enriched information which describes the service well. By simply changing its structural parameters or small programmable parameters, the service can change its behavior significantly. In addition, parameterizing a service saves implementation code, which reduces communication cost when deploying the service. We will discuss other approaches to reduce the code size of the service in next section.

3 An Autonomic Routing Architecture for Sensor Networks

This section presents the node architecture with the proposed autonomic routing service. This architecture is extended from the original design reported in [22]. This section will emphasize the expanded part of the architecture. But for background knowledge, we also briefly describe the design in [22].

3.1 Overview

An autonomic routing service for sensor networks must be lightweight, due to limited available resources. In the proposed autonomic framework, we take three approaches to reduce the size of a routing service, which are shown in Fig. 2.

First, a *shared library* is built to support routing services. The library contains common service operations. Routing modules (packet handlers in Fig. 2) are written by calling the shared library. Since the shared library code will not be loaded until the routing modules start to run [23], routing modules written with the library have smaller code sizes than those without the library. Second, a routing service is *divided* into several handlers, each of which deals with specific category of packets. One example decomposition is to have two handlers; one is to process data packets, while the other processes control packets. This way, part of the routing service can be changed with less cost since only the corresponding handler, instead of the whole service code, needs to be replaced. The third approach is to have the *parameter space maintained separately* by another service (called deployment service in Fig. 2). This way the code for parameter maintenance is saved for the routing service. We will discuss the third approach further with individual architecture components in later sub-sections.

The autonomic architecture includes two parts: a routing service and a deployment service. The gray boxes in Fig. 2 are code that is residing in the node before deploying a routing service, while the white boxes are loadable param-

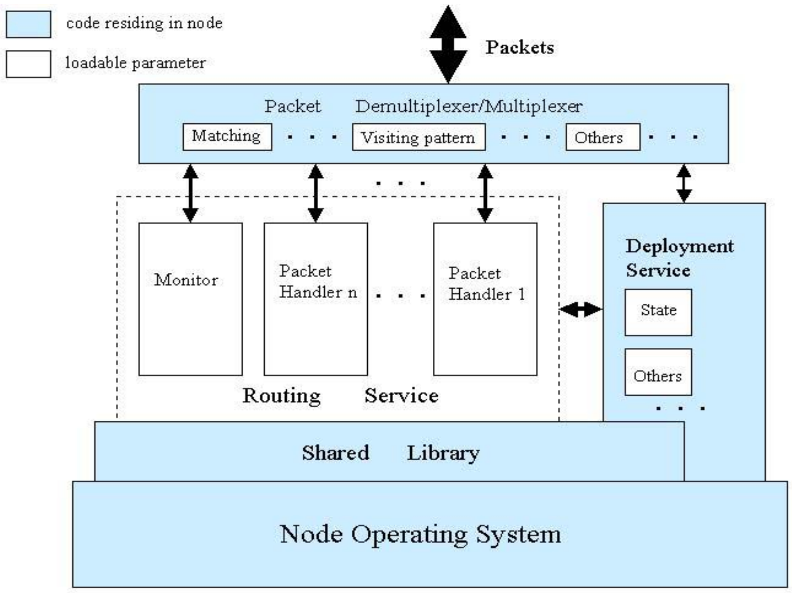

Fig. 2. Sensor Node Architecture with an Autonomic Routing Service

eters. The routing service consists of one or several packet handlers which are deployed by the deployment service. The deployment service also deploys and maintains other parameters (the small white boxes in Fig. 2). The demultiplexer/multiplexer is a logical part of the deployment service and provides networking I/O. The deployment service can also be built partly on top of the shared library if operations are available from the library.

3.2 Deployment Service

The deployment service receives/sends packets (including both parameter packets and routing service packets), processes parameter packets, loads programmable routing components, and configures/maintains parameters. It consists of three logic parts: a packet demultiplexer/multiplexer, a deployment protocol, and a configuring module.

All networking I/O is done via the packet demultiplexer/multiplexer. The routing service controls packet input and output via parameters in the demultiplexer/multiplexer. The first set of these parameters is called *matching* parameters that specify patterns of incoming packets required by packet handlers. The patterns can be packet header fields (such as packet type), packet data, or even dynamic content. A matching parameter can thus be either simply a unit one, a structural one, or even a programmable one. The second set of parameters is the visiting-pattern described in Section 2. The demultiplexer/multiplexer directs input packets by the matching parameters and output packets by the visiting-pattern parameters. This way, a part of job of the routing service is done by the demultiplexer/multiplexer and the service code is reduced. Besides, these parameters are important means to tune the service performance.

The deployment protocol specifies how each sensor node communicates with each other to transfer the parameter space in a network. The basic performance requirement is energy efficiency. This deployment protocol includes two components: an *adaptive flooding* protocol, modified from SPIN (Sensor Protocols for Information via Negotiation [24]), which disseminates parameters throughout a network, and a *synchronization* protocol that enables a sensor node to catch up with its neighbors. In this deployment protocol, a node does not immediately send out data. Instead, it negotiates with neighbors before sending to avoid repeated flooding. Energy is saved in this way. We have found by simulations that negotiations are beneficial when data are large, but are not worthwhile when data are small. In small-data cases, brutal flooding performs better in simulations. Thus the deployment protocol adapts to the brutal flooding when the size of parameter packets are smaller than some threshold.

The reason for the synchronization protocol is that inconsistency among sensor nodes do exist due to engery and communication failure. Besides, it is normal for a sensor node to sleep for a while because of using energy-saving mechanisms such as those in [25], [26], and [27]. After waking up from the sleep, a synchronization operation may be necessary. To start this protocol, a node first broadcasts an *initial request* among its neighbors for each parameter it wants to synchronize (the node can obtain the parameter list via periodic hello messages). The initial request includes the node's version number for the parameter. Upon receiving the request, each neighbor that has a version number greater than the requesting node starts out a timer. The timeout value is used to ensures that the neighbor with the highest version number and remaining energy first times out. After the timeout, the neighbor sends an *initial reply* to the requesting node, suppressing other overhearing nodes. The requesting node also starts a timer immediately after sending the initial request to wait for all possible initial replies from its neighbors. After collecting all initial replies from neighbors, the requesting node chooses the node with highest version number to send a *formal request*, which in turn sends back a *formal reply* to complete the synchronization. If the requesting node does not receive any initial reply before the timeout, it assumes it has updated routing service.

The last component of the deployment service is the core part called configuring module. This configuring module receives parameter packets from the demultiplexer/multiplexer, installs and loads them to bring up the routing service. All parameters of the routing service (including the routing state, and those for the demultiplexer/multiplexer) are configured and maintained by the configuring module. There are two main advantages for maintaining service parameters in a place separate from the routing service. First, as mentioned in Section 3.1, the burden of parameter maintenance is relieved from the routing service, which saves code and deployment cost. Second, managing parameters in a central place has important implications to autonomic computing. The configuring module has all the information of parameters. It also has the privilege to collect routing service packets via the demultiplexer/multiplexer. Thus the configuring module can provide sufficient input to a monitoring mechanism to facilitate performance tuning.

3.3 Routing Service

As shown in Fig. 2, a routing service in the proposed architecture is composed of one or multiple packet handlers. The handlers are transfered to each node via the deployment protocol and are installed by the configuring module. When they are running, the demultiplexer directs packets to the right handlers according to the matching parameters, and the multiplexer sends packets from the routing service to the network according to the visiting-pattern parameters.

State is a special parameter used not only for completing the function of the routing service, but also for monitoring the performance of the service. For example, statistical state can be used to direct performance tuning.

The autonomic routing architecture facilitates parameter changes to tune the service performance. Note that different entities involved in a routing service may have different roles in terms of changing parameters because they have different views on performance metrics. For example, suppose one rough classification of entities includes end nodes, intermediate nodes, and users/administrators. End nodes may obtain metrics of end-to-end delay, intermediate nodes may have better knowledge for local routing paths, while the users/administrators may have a global view. To improve the views of entities (e.g., extending the local views of sensor nodes), some learning techniques such as Markov Chains [28], Bayesian Networks [29], and Neural Networks [30] may be applied. Learning techniques specific to sensor networks are under investigation.

The parameter changes from end-users or administrators can be easily done with the deployment protocol in Section 3.2. To promote parameter changes by node itself, we propose a programmable *monitoring handler* as shown in Fig. 2. The monitoring handler is a special packet handler. It has a specific matching parameter to obtain all required incoming and outgoing routing service packets. The monitor handler has access right to all service parameters.

Since it is built with the routing service, it has knowledge of the semantic meanings of the service parameters and thus can change them to tune the performance. The monitoring handler can have its own parameters such as statistical state and learning functions that may be adjusted by users or system administrators.

4 Implementation

We have implemented a prototype of the proposed autonomic routing framework for sensor networks using C++. The software includes a library, a demultiplexer/multiplexer, a broadcast agent, a synchronization agent, a configuring agent, and a basic class definition of the routing service. The library includes timer functions, packet manipulation functions, network I/O functions, and some tool functions. The demultiplexer/multiplexer, broadcast agent, synchronization agent, and configuring agent compose the deployment service. The basic class defines an interface for the routing service.

The configuring agent maintains a parameter hash table which includes all unit, structural, and programmable parameters of the routing service. There is a generic parameter class from which all parameters are derived. The manipulations of parameters on the hash table, such as "read", "add", "remove", and "update", are also defined in the configuring agent. All modules (including deployment service modules and routing service modules) operate on parameters via the configuring module.

The main advantage of operating parameters via a single delegation (an instance of the configuring agent) is to simplify the synchronization problem when parameters are shared. All operations on parameters are automatically ordered because of the single instance. Thus deadlocks are avoided. There are occasions when synchronizations are necessary. First, among all modules in our software, only packet handlers can write to the state (possibly shared by multiple handlers). Operations involving such writes need to be synchronized. Second, the monitoring handler can write to parameters to tune the performance, and the broadcast and synchronization agents also write to parameters. These operations also need to be synchronized. The generic parameter class in our software thus also includes locking methods that can be used in these situations.

State is a special parameter since it can be shared by packet handlers, and its size can change dynamically during the operation of the routing service. Our approach is to first change a little bit of the syntax for the state parameter. Instead of using concrete values for the "ParameterValue" fields as shown in the example in Section 2, state parameters use type values. Still considering that example, the "ParameterValue" field of the "NeighborID" for the state will be "Number", while the "ParameterValue" field of the "DataType" will be "String", and other fields are changed accordingly. Several types are predefined in our software so that the configuring agent can recognize them. Thus when receiving the state parameter, the configuring agent knows the size of the state and automatically generates "allocate" and "free" methods for the state. This way, packet handlers can dynamically operate on the state parameters.

In our software framework, a routing service (one or multiple packet handlers) is implemented by inheriting from the basic class we defined for a generic service. The routing service must implement a "recv" method so that the demultiplexer knows where to send matched packets. In addition, the routing service must have an instance of the deployment service so that it can operate parameters and send packets via the methods provided by the deployment service. Also, when the routing service is written, the library functions should be used as many as possible. As examples to show how to write routing services using our software, we have implemented the broadcast agent and the synchronization agent as packet handlers that are derived from the basic service class. Implementing these two agents as packet handlers also implies that their parameters can be adapted.

A programmable parameter other than a packet handler is handled differently in our software framework. To have the deployment service recognize these programmable parameter, their names can begin with some special character to differentiate themselves from packet handlers. Our software framework defines a simple basic class for these programmable parameters. The basic class only requires that the constructor obtain an instance of the deployment service so that the state parameters can be accessed. A programmable parameter can be a friend class of the packet handlers that will use it. This way, the internal state/methods of the packet handlers can be shared by the programmable parameter. When the configuring agent receives a programmable parameter, it instantiates the class and then puts a pointer to the instance along with the parameter into the parameter hash table. By this way, the parameter instance will be found and shared when packet handlers want to use the parameter. Only one instance of each programmable parameter is needed for the whole routing service.

The monitoring handler is under developed. We will report the updated implementation in the final paper.

Simulator	ns-2.26
Simulation time	100 seconds
Node number	100
Network size	500 m x 500 m
Radio nominal range	250 m
Transmission cost	600 mw
Reception cost	300 mw
Mata-data size	48 bytes
Parameter size	40, 200, 1000, 5000 bytes

Table 3
Simulation Configurations

5 Simulation Results

We have used ns-2 [31] to obtain preliminary simulation results for our autonomic routing framework. Table 3 lists the simulation configurations. Among the configurations, the meta-data are used by senders in the deployment protocol to negotiate with neighbors before sending actual data. The actual parameter sizes are from 40 bytes and 200 bytes, to 1000 bytes and 5000 bytes. These values reflect sizes for unit, structural, and programmable parameters.

One important metric for our autonomic routing framework is the sizes of the packet handlers since code size affects communication cost in sensor networks. Table 4 shows the sizes of the compiled broadcast and synchronization agents. We can see that the sizes of packet handlers from both agents are much smaller than the library code, with a factor of 10. Note that the compiled code of these two agents includes ns-2 library code. The sizes of the agents are expected to be smaller after removing the ns-2 code (which will be the cases in experiments with actual testbeds).

Fig. 3 shows the delay time and the energy consumption for the broadcast agent. SPIN and brutal-flooding are compared with the proposed adaptive flooding. The delay time (in seconds) at the left side of Fig. 3 is the time needed to disseminate one parameter packet from one node to all other nodes in the network. The values shown on Fig. 3 are averages over 20 deliveries. The delay time for the adaptive flooding is the same as the the brutal flooding when the packet sizes are small (40-bytes and 200-bytes case) since the protocol leverages the brutal-flooding in small-packet cases. The brutal-flooding performs better than SPIN in these cases because the meta-data packets for SPIN have a size comparable to the actual paramter packets while the brutal-flooding does not use the meta-data packets. When parameter packets become

	Code Size	**Code Ratio**
Library code	60460 bytes	1
Broadcast agent (3 handlers)	23468 bytes	0.39
Average size of packet handlers of the BA	7823 bytes	0.13
Synchronization agent (4 handlers)	23480 bytes	0.40
Average size of packet handlers of the SA	5960 bytes	0.10

Table 4
Code Size and Ratio

larger, the adaptive flooding uses a modified version of SPIN. Both the adaptive flooding and SPIN have bigger delays than the brutal flooding. But since brutal flooding involves more collisions in large-packet cases, the delay differences between brutal flooding and the adaptive flooding become smaller than the small-packet cases.

The right side of Fig. 3 shows the comparisons on energy consumption. The values are average energy dissipated (in Joules) per node in each delivery. The adaptive flooding and brutal flooding perform better in small-packets case again because they don't use meta-data. In large-packet cases the adaptive flooding and SPIN performs much better (in a factor of 3 in 5000-bytes case) because they use small meta-data packets to negotiate before sending actual big parameter. Fig. 3 shows that parameters can be disseminated in an energy-saving way with an acceptable delivery time.

Note that the values of the adaptive flooding are different from the SPIN. This is because the adaptive flooding uses a different timeout mechanism that considers more on energy efficiency. With this timeout mechanism, the adaptive flooding saves energy slightly more than SPIN. Refer to [22] for details.

Fig. 4 shows the delay time and the energy consumption for the synchronization agent. To obtain the test results, the simulation time is divided into periods, in each of which one node is randomly picked to sleep and wake up. When the node wakes up, it carries out the synchronization protocol. The values in Fig. 4 are averages over 20 synchronization operations. The figures also shows the results of the one-hop broadcast for comparisons.

When parameters are small, the one-hop broadcast performs well for the same reason of the broadcast agent. Our synchronization agent uses the one-hop broadcast in this case. When packets become large, the delays of the synchro-

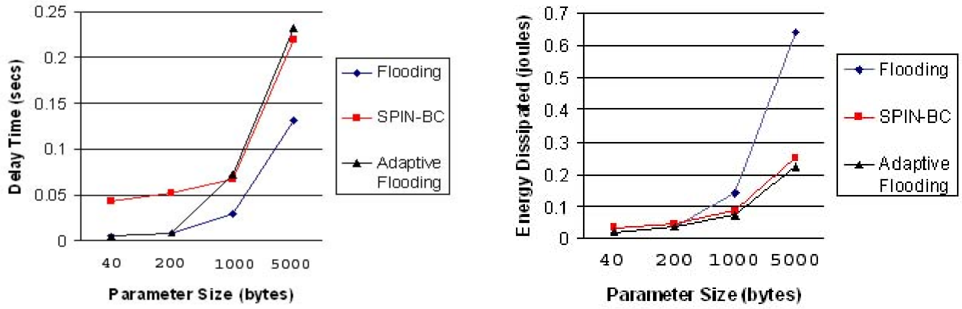

Fig. 3. Broadcast Agent Performance

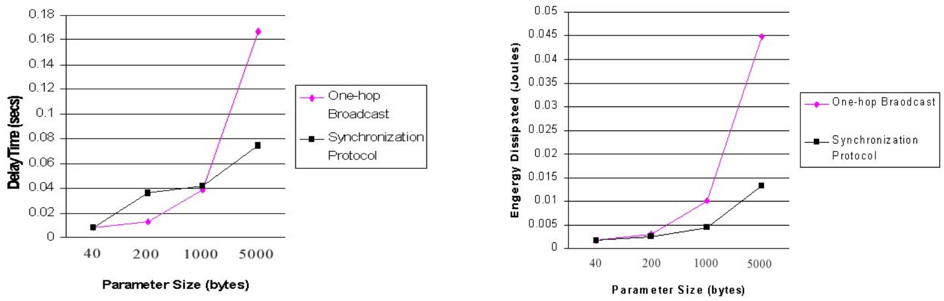

Fig. 4. Synchronization Agent Performance

nization protocol increase in a ratio much lower than the one-hop protocol (for example, in Fig. 4, the delay of the former in case of 5000 bytes is smaller than the delay of the latter with a factor of 2). This is because there is exactly one neighbor that sends the big parameter under the synchronization protocol, while all neighbors that have a higher version send the parameter under the one-hop broadcast case. The delays become more stable in the proposed synchronization protocol.

Similar to the case of the broadcast agent, the energy efficiency of the synchronization agent is obvious in Fig. 4; the merit increases when packets become larger.

Currently our deployment protocol uses only a simple adaptive algorithm. The adaptive mechanism should be a functional place for a monitoring handler. We will implement the monitor handler to experiment other algorithms.

6 Related Work

Active networks ([32] and [33]) allow user code to be executed when data packets arrive at routers. The code is either carried in the packets or dynamically loaded from code servers. In a sensor network conforming to the proposed autonomic routing framework, data packets do not carry user code and there is no code loading upon the arrival of the data packets. The routing service code specific to the application is installed and configured via parameter packets and has been in place when data packets arrive. Parameter packets appear in the network only when the deployment protocol is used. Given that a routing service processes data packets most of the time, the number of parameter packets is much lower than the number of data packets. Thus it is not an energy-saving way in sensor networks to have data packets carry code. Code loading in the autonomic framework is less flexible than active networks, but meets well the requirements of sensor networks. Active networks do not address how to adapt loaded services, while this issue is studied thoroughly in the autonomic routing framework under the context of sensor networks.

Scout [34], Router-Plugins [35], Click [36] and Internet Exchangeable Architecture [37] study extensible router architectures [38] to meet the needs of sophisticated services in the Internet. These routers follow powerful and sometimes heavyweight models and are not exactly applicable to sensor networks where resources are very constrained. Besides, it is not easy to deploy extensible routers on the Internet due to their significant changes to traditional routers that do not have an open architecture. As a contrast, sensor network architectures are under development, and the deployment of autonomic routing framework is relatively easy.

Filter-based architecture in sensor networks ([39] and [40]) specifies a software structure that contains a list of filter handlers, each of which is executed when its attributes match incoming packets. This architecture provides a flexible way to add application-specific code into sensor nodes in the form of filters [41]. One good example of such application-specific filters is the information-driven tracking filter ([42] and [43]) that uses tracking information to direct packets and conserve energy. The autonomic routing framework are more flexible than the filter-based architecture. First, all filters in the filter-based architecture are statically pre-configured into nodes. Routing services in this architecture are not parameterized, and thus can not adapt significantly. Second, the core part of the filter-based architecture is based on Directed-diffusion [44] and can not be changed by users. Thus it is impossible to implement other significantly different routing services such as TTDD on top of the core part. With our autonomic framework, any routing service can be supported.

In [45], each sensor node contains a virtual machine with a small instruction set. Each instruction in the set abstracts some high-level function carried out by TinyOS [1]. One important instruction is to support code migration in a network. Services constructed via this instruction set are expected to have small size and can be loaded in a memory-and-energy-saving manner. This virtual machine could be a potential environment to implement routing services. However, the expressiveness of the instruction set is still not clear. As shown in [46], it may be difficult to write significant routing services for sensor networks with the virtual machine's ultra-compact instruction set.

Sensorware ([47] and [46]) is another programmable environment but provides a higher-level service abstraction than the virtual machine described above. Services then can have smaller code size, written using a script language that can be understood by the sensorware built in each node. The sensorware targets to support a specific kind of application called localized distributed computation such as a collaborative signal processing task. These applications generally only involve a subset of sensor nodes and are transient, which are different from the target of the proposed autonomic routing framework. Sensorware needs an underline routing service to work. In terms of active networks, the sensorware can be regarded as "user-level" active applications, while the autonomic routing service can be considered as a "system-level" active application in sensor networks.

7 Conclusions and Future Work

We have described an autonomic routing framework for sensor networks. This framework is established on a well-defined parameter space. We believe that the proposed parameterization has a general guidance to build other autonomic services as well. To meet the resource-scarce property of sensor networks, we have proposed three approaches, shared library, decomposition, and separate parameter management, to build a lightweight routing service. The deployment service in our framework can automatically deploy the routing service throughout the network in an energy-efficient way, and manage parameters in a central space to simplify synchronization problems. A monitoring handler accompanying with the routing service has been proposed to tune service parameters by leveraging the deployment module. We have presented details for a prototype implementation. The simulation results have shown that significant routing services can be written with relatively small sizes and the deployment protocol is energy-saving with satisfiable delays.

We plan to report the implementation for the monitoring handler in the final paper. Self-protection mechanisms of this autonomic framework will also be explored. We will carry out experiments on a test bed with sensor nodes conforming to this autonomic routing framework.

References

[1] Jason Hill and et al. System architecture directions for networked sensors. *ASPLOS*, November 2000.

[2] K. Pister and et al. Smart dust: wireless networks of milimeter-scale sensor nodes. *Highlight Article in 1999 Electronics Research Laboratory Research Summary*, 1999.

[3] G. Pottie and et al. Wireless sensor networks. *Communications of the ACM*, 2000.

[4] D. Estrin, R. Govindan, and et al. Next century challenges: scalable coordination in sensor networks. *Mobicom*, August 1999.

[5] D. Estrin, L. Girod, and et al. Instrumenting the world with wireless sensor networks. *ICASSP*, May 2001.

[6] J. M. Kahn and et al. Mobile networking for smart dust. *Mobicom*, August 1999.

[7] J. Cerpa, D. Estrin, and et al. Habitat monitoring: application driver for wireless communications technology. *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, April 2001.

[8] L. Girod, D. Estrin, and et al. Robust range estimation using acoustic and multimodal sensing. *IEEE Conference on Intelligent Robots and Systems*, 2001.

[9] Brad Karp and H. T. Kung. Gpsr: Greedy perimeter stateless routing for wireless networks. *Mobicom*, 2000.

[10] Yan Yu, Ramesh Govindan, and Deborah Estrin. Geographical and energy aware routing: a recursive data dissmination protocol for wireless sensor networks. *UCLA CS Tech. Report*, 2001.

[11] Dragos Niculescu and Badri Nath. Trajectory based forwarding and its applications. *Mobicom*, September 2003.

[12] Badri Nath and Dragos Niculescu. Routing on a curve. *Hotnets I*, October 2002.

[13] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalabel and robust communication paradigm for sensor networks. *Mobicom*, 2000.

[14] Fan Ye, Haiyun Luo, Jerry Cheng, Songwu Lu, and Lixia Zhang. A two-tier data dissemination model for large-scale wireless sensor networks. *Mobicom*, September 2002.

[15] Hu Zhou and Suresh Singh. Content-based multicast for mobile ad hoc networks. *Mobihoc*, August 2000.

[16] David Braginsky and Deborah Estrin. Rumor routing algorithm for sensor networks. *WSNA*, September 2002.

[17] J. Moy. Rfc 2328 - ospf version 2. *Request for Comments*, April 1998.

[18] C. Hedrick. Rfc 1058 - routing information protocol. *Request for Comments*, June 1988.

[19] Samuel R. Madden, Robert Szewczyk, Michael J. Franklin, and David Culler. Supporting aggregate queries over ad-hoc wireless sensor networks. *WMCSA*, June 2002.

[20] Alan Ganek and et al. Autonomic computing: Ibm's perspective on the state of information technology. *http://www.research.ibm.com/autonomic/manifesto/*, October 2001.

[21] Tim Bray and et al. Extensible markup language (xml) 1.0 (second edition). *http://www.w3.org/TR/REC-xml*, October 2000.

[22] Yu He, C. S. Raghavendra, Steven Berson, and Robert Braden. A programmable routing framework for autonomic sensor networks. *Active Middleware Service (AMS)*, June 2003.

[23] David A. Wheeler. Program library howto. *http://www.tldp.org/HOWTO/Program-Library-HOWTO/*, December 2002.

[24] W. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. *Mobicom*, August 1999.

[25] S. Singh and C. S. Raghavendra. Pamas - power aware multi-access protocol with signalling for ad hoc networks. *ACM Computer Communication Review*, July 1998.

[26] Ya Xu, John Heidemann, and Deborah Estrin. Geography-informed energy conservation for ad hoc routing. *Mobicom*, 2001.

[27] Wei Ye, John Heidemann, and Deborah Estrin. An energy-efficient mac protocol for wireless sensor networks. *Infocom*, June 2002.

[28] E. Behrends. Introduction to marcov chains. *Vieweg*, 1999.

[29] F. V. Jensen. An introduction to bayesian networks. *UCL Press*, 1996.

[30] K. Gurney. An introduction to neural networks. *Routledge*, 2002.

[31] USC/ISI. Network simulator. *http://www.isi.edu/nsnam/ns/*.

[32] D. L. Tenenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, April 1996.

[33] D. L. Tenenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. Minden. A survey of active network research. *IEEE Communication Magazine*, January 1997.

[34] L. Peterson and et al. An os interface for active routers. *IEEE J-SAC*, 19(3):473–487, March 2001.

[35] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: A software architecture for next generation routers. *IEEE/ACM Transactions on Networking*, 8(1):2–15, Februray 2000.

[36] E. Kohler and et al. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[37] Intel Corporation. Ixp1200 network processor datasheet. September 2000.

[38] Larry Peterson Yitzchak Gottlieb. A comparative study of extensible routers. *IEEE OPENARCH*, June 2002.

[39] John Heidemann, Fabio Silva, and et al. Building efficient wireless sensor networks with low-level naming. *SOSP*, October 2001.

[40] John Heidemann, Fabio Silva, and et al. Diffusion filters as a flexible architecture for event notification in wireless sensor networks. *USC/ISI Tech. Report*, April 2002.

[41] Fabio Silva, John Heidemann, and et al. Network routing application programmer's interface (api) and walk through 9.0.1. *http://www.isi.edu/scadds/papers/*, December 2002.

[42] Feng Zhao, Jaewon Shin, and et al. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Singal Processing Magazine*, March 2002.

[43] M. Chu, H. Haussecker, and et al. Scalable information-driven sensor querying and routing for ad hoc heterogeneous sensor networks. *Int'l J. High Performance Computing Applications*, 16(3), Fall 2002.

[44] Fabio Silva and et al. Diffusion 3.1.3 code. *http://www.isi.edu/scadds/software/*, July 2002.

[45] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. *ASPLOS X*, October 2002.

[46] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. *MobiSys*, May 2003.

[47] Athanassios Boulis and Mani B. Srivastava. A framework for efficient and programmable sensor networks. *OPENARCH*, July 2002.