Hood: A Neighborhood Abstraction for Sensor Networks

Kamin Whitehouse Cory Sharp Eric Brewer David Culler

{cssharp,kamin,brewer,culler}@cs.berkeley.edu
Department of Electrical Engineering and Computer Science
University of California, Berkeley
Berkeley, CA 94720-1776

ABSTRACT

This paper proposes a neighborhood programming abstraction for sensor networks, wherein a node can identify a subset of nodes around it by a variety of criteria and share state with those nodes. This abstraction allows developers to design distributed algorithms in terms of the neighborhood abstraction itself, instead of decomposing them into component parts such as messaging protocols, data caches, and neighbor lists. In those applications that are already neighborhood-based, this abstraction is shown to facilitate good application design and to reduce algorithmic complexity, inter-component coupling, and total lines of code. The abstraction as defined here has been successfully used to implement several complex applications and is shown to capture the essence of many more existing distributed sensor network algorithms.

1. INTRODUCTION

One landmark paper on the design of sensor networks foresaw the prevalent use of *localized* algorithms: *"a distributed computation in which sensor nodes only communicate with sensors within some neighborhood"* [6]. Indeed, many distributed sensor network algorithms are based on some concept of a neighborhood; each node selects some set of important neighbors and maintains state about each of them. However, the neighborhood is still not a *programming primitive* in the sensor network community. Neighborhood-based algorithms are decomposed into other components, such as neighbor lists, data caches, and messaging protocols, all of which are components of neighborhoods but are not treated uniformly as a single abstraction. This is partly because each neighborhood is slightly different; different types of nodes must be selected and different state must be maintained about them. Developers are still grappling for a clear, solid abstraction that defines the ideas and relationships common across neighborhoods.

The contribution of this paper is to introduce a new way of thinking about the relationship between several fundamental concepts of neighborhoods: *membership*, *data sharing*, *data caching*, and *messaging*. This relationship is solidified in a single unified programming primitive called ***Hood***, which allow developers to think about algorithms directly in terms of neighborhoods and data sharing instead of decomposing them into messaging protocols, data caches, and neighbor lists.

A neighborhood in Hood is defined by a set of criteria for choosing neighbors and a set of variables to be shared. A node can define multiple neighborhoods with different variables shared over each of them. For example, Hood can define a one-hop neighborhood over which light readings are shared and a two-hop neighborhood over which both locations and temperature are shared. Once the neighborhoods are defined, Hood provides an interface to read the names and shared values of each neighbor. Beneath this interface, Hood is managing discovery and data sharing, hiding the complexity of the membership lists, data caches, and messaging.

The proposal of programming abstractions like this one is notoriously difficult to quantify and evaluate, but this paper makes a series of three arguments to support Hood. First, neighborhood-based algorithms are easier to design, implement, and modify with Hood than with only traditional primitives. This is shown by a case study of one application that was designed using both methodologies. Second, the quality of the implementation using Hood is better, as shown by a comparative analysis of the two implementations. Third, the conception of Hood captures the essence of many distributed sensor network algorithms, as shown by an analysis of the different usages of neighborhood concepts seen in sixteen existing applications.

Section 2 describes the abstraction provided by Hood. Section 3 describes the implementation of Hood and its limitations while providing a brief introduction to TinyOS, nesC and Active Messages. Section 4 presents an object tracking application and its original implementation without using Hood. Section 5 shows the second implementation of object tracking which does incorporate Hood. Section 6 analyzes the two implementations in terms of state machine complexity, inter-component coupling, and lines of code. Section 7 examines existing neighborhood implementations and shows

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
MobiSYS'04, June 6–9, 2004, Boston, Massachusetts, USA.
Copyright 2004 ACM 1-58113-793-1/04/0006 ...$5.00.

that Hood can capture these usages. It also presents another application that was implemented with Hood to show that it can support sophisticated distributed algorithms. Section 8 describes how the simple concepts introduced here can be extended to support more complex neighborhoods such as multi-hop or bi-directional neighborhoods. Section 9 presents other programming abstractions and their relationships with Hood. Finally, Section 10 supplies a summary and conclusions.

2. THE HOOD ABSTRACTION

Hood provides an abstraction for a very common behavior in sensor network algorithms: sharing and viewing attributes of neighboring nodes. For example, a node might want to view the locations of its nearest nodes or the light values of neighbors in the shade. This behavior involves a unique relationship between the concepts of *data sharing* and *neighborhoods*; data sharing is limited to the scope of the local neighborhood, and the neighborhood is defined by its membership and the data being shared. The goal is to capture useful, lightweight mechanisms that match the essence of many distributed algorithms in wireless sensor networks.

Hood captures this relationship by establishing 1) which attributes are to be shared, and 2) the membership criteria that characterize a neighborhood. Once defined, Hood provides an interface to list the names of the current neighbors and to view their shared attributes. Beneath this interface, Hood is automatically discovering neighbors and caching the values of their attributes while simultaneously sharing the values of the node's own attributes.

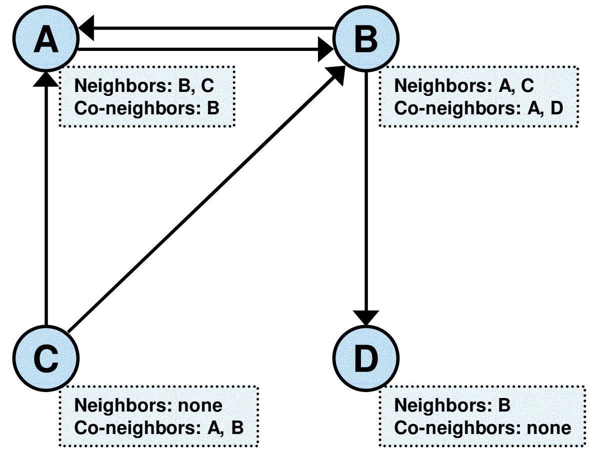

Figure 1: Neighbor membership. A node locally caches readings and data of its *neighbors*; A records data reported by B and C. A node's own readings and data are remotely cached at its *co-neighbors*; A's data reports are recorded at B. A node only maintains a neighbor membership list and has no information of its co-neighbor membership; A does not know that its data reports are cached at B. This inherent asymmetry is made possible by exploiting a cheap broadcast mechanism. Symmetry may at times occur; A and B record each other's data – though, this symmetric relationship is incidental.

2.1 Broadcasting and Filtering

Key to the implementation of Hood is the ***broadcast/filter*** mechanism used for both data sharing and neighborhood discovery. When attributes are shared they are always broadcast. The receiving nodes "filter" the incoming attributes to determine which nodes are adequate neighbors and which of their attributes should be cached. This decouples the *owner* of an attribute from the *observers* of the attribute. The receiver of an attribute has exclusive control to add and cache a node in its neighbor list; the owner of the attribute has neither control over nor knowledge about the nodes that are actively recording its state. This means that in Hood a neighborhood is fundamentally a local construction at each and every node, in contrast to *groups* where membership is shared an manipulated among the nodes in the group.

A node typically only caches attributes from nodes that it considers valuable. To do so, the node may need to define multiple neighborhoods, each with its own definition of a valuable neighbor and each caching different types of attributes. For example, a node might define a routing neighborhood that caches the location information of valuable routing nodes and a sensing neighborhood that caches sensor information of valuable sensing nodes. Each node's neighborhood independently decides which neighbors are valuable based on the shared attributes they are broadcasting, fills its neighbor list, and caches the attributes it deems important. The decoupling of owners from observers allows this to happen without the owner knowing who deems it to be valuable for routing or sensing.

Implicit in the decoupling of owners and observers is a potential for *asymmetry* in neighborhoods. For instance, in Figure 1, B is in D's neighborhood but D is not in B's neighborhood. To help reason with this relationship, we introduce the notions of a neighbor and co-neighbor. A node caches data reports from its ***neighbors***, and a node's own data reports are cached at its ***co-neighbors***.

The broadcast/filter process is part of what makes Hood especially suitable for sensor networks. First, it allows Hood to exploit the cheap broadcast channel inherent in wireless networks. Further, it promises only the weak sharing semantics that unreliable, low-bandwidth networks can provide; the neighbors in the neighbor list and the cached values of a neighbors' attributes represent only the best or most-recently observed. Any stronger guarantees about consistency, coherence or reliability are intentionally deferred to the application level.

2.2 Basic Concepts

Attributes define the elements of a node's state that are shared with its neighbors, such as sensor readings or geographic location. When a node updates its own attribute, the value is said to be *reflected* to its co-neighbors, much like traditional reflective memory (RM) [15]. Exactly how data is reflected is determined by the ***push policy***. Typically, this is simply to broadcast the value each time it is set. An alternative might be to broadcast periodically, reliably, or never at all. The latter case might be useful for allowing application-level control over attribute sharing, which is possible through the push/pull interface described in Section 3.

When an attribute is received at a co-neighbor, it is passed through the filters of each neighborhood defined on that

```
interface Attribute {
  command int get();
  command void set( type value );
  command void push();
  event void updated( type value );
}

interface Hood {
  command nodeID[] getNeighbors();
  command bool isNeighbor( nodeID id );
  event void removingNeighbor( nodeID id );
  event void addedNeighbor( nodeID id );
  command void bootstrap();
}

interface Reflection {
  command int get( nodeID id );
  command void pull( nodeID id );
  event void updated( nodeID id, type value );
}

interface Scribble {
  command int get( nodeID id );
  command void set( nodeID id, type value );
  event void updated( nodeID id, type value );
}
```

Figure 2: Hood attribute, neighborhood, reflection, and scribble programming interfaces.

node. ***Filters*** examine each shared attribute to determine which nodes are valuable enough to place in the ***neighbor list*** and which attributes of those nodes need to be cached. For each node in the neighbor list, a ***mirror*** is allocated, which represents the local view of that neighbor's state. It contains both ***reflections***, which are cached versions of that neighbor's attributes, and ***scribbles***, which are local annotations about that neighbor. Scribbles are often used to represent locally derived values of a neighbor such as a distance estimate or link-quality estimate.

3. IMPLEMENTATION

Hood was developed in nesC on the TinyOS software platform [8], which provides a component-based software model and an Active Message communication model. nesC *modules* are software components that are *wired* together to form an application, much like hardware components on a schematic. Components are described using bidirectional *interfaces*, where *providers* of an interface process its *commands* and *users* of an interface handle its *events*. A module that uses an interface is *wired* to a module that provides that interface using a *configuration* file. Active Messages is a messaging abstraction where each message type is associated with a unique block of code that is run each time that type of message is received.

Hood provides individual interfaces for attributes, neighborhoods, reflections, and scribbles, shown in Figure 2. The architecture between these interfaces is shown in Figure 5; Figure 4 assists in understanding the dependencies expressed in the diagram.

The rest of this section describes how these interfaces and supporting modules are automatically generated, how Hood caches, sends, and receives data from other nodes, and finally some costs and limitations of both the abstraction and the implementation.

```
generate attribute LightAttribute from int;

generate neighborhood LightHood {
  wire filter LightThreshold;
  set max_neighbors to 5;
  reflection LightRefl from LightAttr;
}
```

Figure 3: Sample specifications that generate underlying code for Hood.

3.1 Code Generation

One of the main challenges in designing Hood is that each neighborhood requires slightly different algorithms and data structures, which are difficult to parameterize. To address this, we developed a form of code generation to allow parameterization of each neighborhood implementation.[1]

Some nesC configurations of Hood are created with `generate` commands that parameterize the underlying module with constants, algorithms, and data structures. One aspect of this technique is that the algorithmic parameters take the form of a user supplied nesC component. This minimizes the weight of the Hood abstraction by providing a foundation and framework for neighborhoods without needlessly restricting the developer. Furthermore, in this scheme, the source template for the generated code can also be specified as a parameter, allowing the user to change any aspect of Hood.

As an example, the `LightHood` specification in Figure 3 specifies the `filter` algorithm of the neighborhood to be the `LightThreshold` component, sets the `max_neighbors` to 5, and derives the data structure and wiring necessary for the `LightRefl` reflection from the `LightAttr` attribute.

This code generation technique is not specific to Hood, and we have developed it as a general architecture to support other services we use in our sensor networks. For instance, `generate` specifications can be embedded within each other, so one algorithmic parameter may specify a module which is itself generated.

3.2 Caching Data

Hood uses neighborhood and attribute modules to cache its data. At the core of a neighborhood component is an array of mirrors, where each mirror is a structure that holds the `nodeID`, reflections, and scribbles for a neighbor. The cache is statically allocated to hold the maximum number of neighbors specified for the neighborhood. The neighborhood component provides one interface to access the membership list and individual interfaces for each scribble and reflection. The neighborhood configuration also automatically wires to its specified filter component and to a `NeighborhoodComm` module, which is a data marshalling layer that runs over the standard radio stack.

A special attribute component is used to cache the value of each attribute. The attribute configuration automatically wires to its push policy and to the `NeighborhoodComm` module. If the `auto-push` parameter is set, which is the default, then the attribute reports changes to its value using `NeighborhoodComm`.

Hood statically allocates the maximum amount of mem-

[1]The syntax presented here is simplified for expositional purposes.

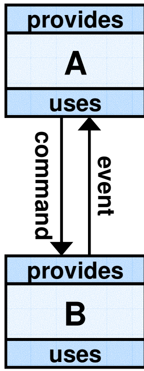

Figure 4: This diagram describes how to interpret the connections between modules in Figure 5. Modules A and B each *provide* and *use* some interfaces. Any arrow connecting the bottom of A to the top of B means that A uses an interface provided by B. An arrow pointing from A to B describes a *command*: behaviors that A can invoke on B. An arrow pointing from B to A describes an *event*: behaviors that B can invoke on A.

ory required for its data caches, because we require that a neighborhood is always able to maintain state for its maximum number of neighbors. This means that memory remains allocated for empty or partially-empty neighborhoods and for attributes that have not yet been reflected. Local reflections of the same attribute are also stored once in each relevant neighborhood. While regrettable, this is the only way to guarantee memory availability for all attributes and neighbors. A misguided attempt to unify the data caches by storing each reflection only once incurs the cost of reference counting and other additional problems for little or no benefit.

3.3 Sending Data

The *push policy* component allows the user to provide different sharing semantics for an attribute beyond the simple auto-push. For example, a policy might be to push the value periodically in mobile networks where neighborhoods change frequently. For important changes, the policy may push the attributes several times for robustness to packet loss. Each time the attribute is written, the *push policy* receives notification that the value has been updated and determines what action to take. Because the push policy is a standard nesC component, any policy can be written by the user.

A more complicated push policy may be required for an attribute that is shared over multiple neighborhoods each with different update requirements. For example, a `Location` attribute may be shared over both a localization neighborhood, which needs all possible updates of neighbor locations, and a geographic routing neighborhood, which only needs large changes but sent robustly. It is left to the application developer to ensure that the sharing policy meets the requirements for all relevant neighborhoods. This can be guaranteed, though perhaps suboptimally, by specifying multiple relevant push policies for the attribute.

A push policy can also be used to aggregate several attributes into one update message. This is necessary if two attributes must be guaranteed to be consistent with each other, such as light value and geographic location. This is also especially useful for certain classes of membership filters, discussed below.

3.4 Receiving Data

When a remote attribute update is received through `NeighborhoodComm`, it is passed into the filter of each neighborhood that is reflecting it. The filter may 1) update

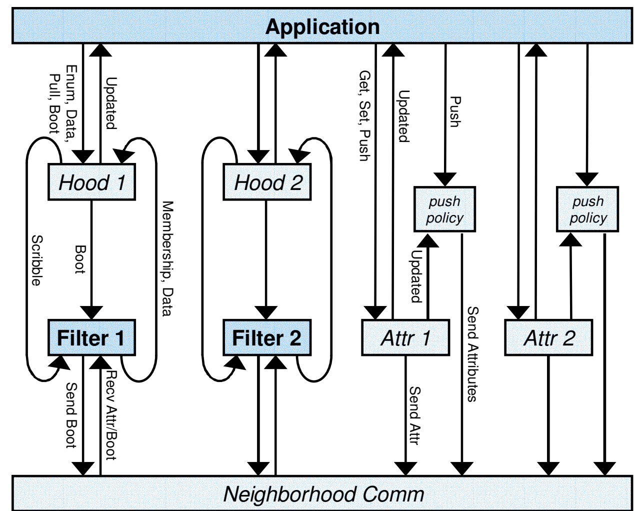

Figure 5: This illustrates a component model for an application with two neighborhoods and two shared attributes. For each neighborhood, a Hood component is created and wired to its respective filter. For each attribute, an Attribute component is created and is wired to its respective push policy. Attributes are sent over the radio through NeighborhoodComm by Attribute components and received by Filter components.

the neighbor mirror if it is already a member, 2) remove the neighbor if it no longer meets membership criteria, or 3) add the neighbor if it meets membership criteria, possibly first ejecting a worse neighbor if the neighbor list is full. In this way, discovery is a natural consequence of the push/filter process; the neighbor list and reflections are populated simultaneously. This technique of locally isolating the management and knowledge of membership is what allows Hood to embrace asymmetry and exploit the cheap broadcast mechanism inherent in wireless networks.

Writing to a scribble goes through the neighborhood filter, as well, though it originates from the neighborhood component instead of from the `NeighborhoodComm` component. This is necessary for the cases in which a scribble, such as a link-quality estimate, is a significant factor of neighbor membership.

Some filters may require more than one attribute to evaluate a neighbor. For example, a light neighborhood may need both light value and geographic location to decide if a node should gain membership. This is an example of when it is useful to use the push-policy to aggregate several attributes into the same packet.

If neighboring nodes are not actively pushing attributes, then the neighbor list and reflections remain unpopulated. This can be a problem if, for instance, a node joins an area of the network where nodes infrequently share their attributes. For such cases, the `bootstrap` command sends a request to all potential neighbors to update their attributes relevant to membership. The filter module must implement `bootstrap` because it is the only component that knows which attributes are important. `bootstrap` is similar to pulling a

reflection from a neighbor, except that it requests data from all nodes, not just one.

3.5 Costs and Limitations

A potential issue with Hood is that it hides the messaging cost from the application writer; a simple write into a shared variable can cause multiple messages to be sent. While explicitly separating shared variables into special components is an effort to reveal this to some extent, the exact cost in not apparent. For example, Hood is only well suited for platforms with a cheap broadcast mechanism. If one is not available, perhaps because of a TDMA communication protocol such as that used in Bluetooth, sharing variables is no longer cheap, instead requiring a separate message to be sent for each neighbor. Much for the reason of limiting hidden costs, Hood makes no guarantees toward consistency and reliability of attribute sharing, and the cost of dropped packets is explicitly deferred to the push policy designer. In this way, Hood imposes no extra or unexpected costs due to dropped packets or network partitions.

Because it does not impose constraints on the application design beyond those implicit in the abstraction, there are some potential pitfalls when using Hood. For instance, allocating too little buffer space or using malformed filters could result in membership thrashing or membership race conditions in which the same nodes are continually replacing each other in the neighbor list. Updating an attribute based on the reflection of another node could cause livelock or infinite recursion, where each push from one node causes many pushes from other nodes repeatedly. Finally, a single event in a sensor network, such as a moving object, might cause a set of nodes to simultaneously push attribute updates, causing network collisions. This has been ameliorated in the past by using push policies that inject a random delay before pushing.

Establishing membership may be difficult if a filter simultaneously requires both a scribble *and* an attribute. For example, a geographic routing neighborhood may select nodes closer to the routing destination and with the highest radio link quality. Hood has no mechanism to directly support this because the two pieces of data are coming from different sources; the location attributes originate at the remote nodes but link-quality estimates originate at the local node. One solution is to use two neighborhoods: a *candidate* neighborhood which maintains link quality estimates, and a full routing neighborhood which establishes membership in part based on the candidate neighborhood.

Consistency *within* a node between local caches of the same attribute is guaranteed because the only writers to reflections are the filters, which are always updated together from `NeighborhoodComm`, a service provided by the Hood library. Hood, on the other hand, does not to employ cache consistency techniques *between* remote nodes, such as those techniques used in more traditional DSM architectures [4]. Because a node does not explicitly know who its co-neighbors are, it cannot guarantee that its co-neighbors receive its attribute updates. This artifact is a consequence of the CAP principle [7]: it is impossible to have both consistency and availability in the presence of network partitions. Because we assume that sensor networks are almost always partitioned to some extent, we design Hood to provide data availability over consistency.

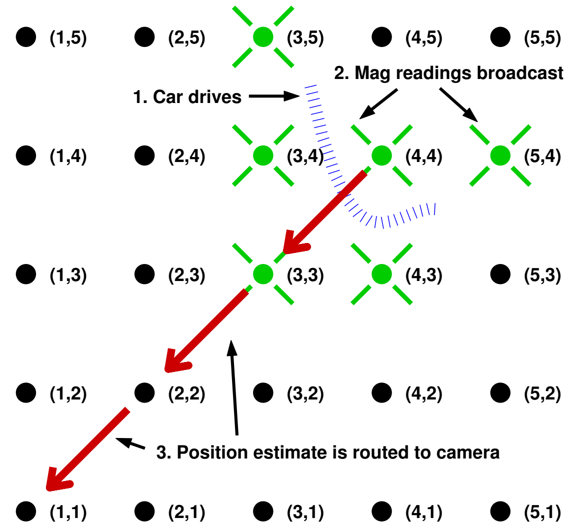

Figure 6: Event Sequence in OTA. 1) The car drives around position (4,4) (dashed-line) 2) six nodes broadcast readings (lightened nodes) 3) node (4,4) declares itself the leader, aggregates the readings and routes them to the base station (dark arrows).

4. OBJECT TRACKING

The object tracking application (OTA) is a distributed sensor network application that detects and reports the position of a moving object within a sensor field. It is also the largest TinyOS application known to date, and in fact first motivated the abstraction of the neighborhood concept as presented in this paper. As a case study of the Hood abstraction, this section describes the design of OTA by the authors before the idea of Hood was well developed. Section 5 will present a later design of OTA which incorporates Hood. At the time of the first implementation, the developers were familiar with the programming primitives provided by TinyOS, but had an imprecise notion of neighborhoods and data sharing.

In the implementation discussed here, a remote control car is driven through a field of magnetometers and the network routes a position estimate of the car to a camera, which visually follows the vehicle. Figure 7 (left) is a photograph of the final application in action. Figure 7 (right) is an extension of the application in which the camera is replaced by an autonomous robot which chases the moving object. When no objects are moving in the sensor field, all nodes silently monitor their magnetometers. All nodes perform some form of localization and report their locations with their neighbors. When a node detects a magnetic disturbance, it also reports this to it neighbors. Each time instance, the node with the largest disturbance declares itself the leader, estimates the location of the disturbance and routes that estimate to the camera. The event sequence of OTA is depicted in Figure 6.

The programming primitives of TinyOS encourage the distributed algorithm above to be broken down in terms of software components and messaging protocols: each data type is to be shared over a different messaging protocol, and each messaging protocol is to be implemented by a different soft-

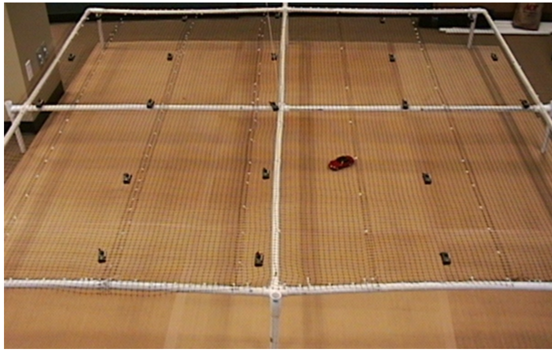

Figure 7: Object tracking applications. left) indoor testbed with 25 nodes, a RC car and a pan/tilt camera. As the remote control car is driven through the sensor field, the network locates the car and send the position to the camera, which visually tracks the vehicle. right) real outdoor deployment with 100 nodes, and two robots. As one robot moves, the network reports its position to the second robot (with the safety cone), which pursues it autonomously.

ware component. This naturally leads to components that store an attribute, share that attribute with neighboring nodes, and cache neighbors' attribute values. Such components are shown in Figure 8.a. This type of design is composable in that each component is self-contained and, as such, can be removed or replaced without affecting any other components. Following this rationale, the OTA developers originally decomposed the application into three components, *tracking*, *routing*, *localization* which stored and shared magnetometer values, link-quality estimates, and geographic locations, respectively. This software architecture is described below.

- The **localization** component stores the node's location and reports a location message whenever it changes. When a location message is received, the value is cached in an array with the neighbor's ID, if there is space. It provides the location of remote nodes to other components through a simple read-value interface to its data cache.

- The **tracking** component monitors the magnetometer and reports a sensing message when it significantly changes. When a new location is learned in the location neighborhood, that node is evaluated for being in the tracking neighborhood as well. When a sensing message is received, the value is stored in an array with the neighbor's ID if that neighbor is in the neighbor list. If the node is the leader, it calculates a position estimate using the local sensing cache and locations from the localization cache interface. It sends an estimation message through the routing component for delivery to the camera node.

- The **routing** component uses geographic routing. When a new location is learned in the location neighborhood, that node is evaluated for being in the routing neighborhood as well. Whenever it receives an estimation message, either from the tracking component or from another node, it decides which node is closest to the destination by searching through the locations of its neighbors via the localization cache interface.

This initially feels like a good decomposition because each module is specifically responsible only for the messaging, caching, and neighbor list directly related to a certain data type. The design is flawed, however, in that the membership requirements are not independent. The tracking and routing components need the location of each node in their neighbor lists but the localization component populates the location cache using its own membership criteria, *irrespective of which locations the other components need*. Therefore, the API to the location component must be extended to allow the coordination of the location cache and location messages with routing and tracking membership. The resulting architecture is illustrated in Figure 8.b.

In this architecture, the designers chose to couple data sharing and data caching because they share the same messaging protocol and are therefore easy to implement in the same component. The problem is that each component in the system then requires conflicting membership criteria from a single data cache. An alternative implementation might have been to couple caching with membership instead of with data sharing. In OTA, that would mean storing the location of each node redundantly in the tracking or routing neighborhoods to keep the location cache consistent with the other two neighbor lists, as shown in Figure 8.c. The problem with this implementation is that tracking and routing must then share a messaging protocol with the localization component; instead of being coupled through an interface, they are coupled through a messaging protocol.

It seems inevitable that all three software components be highly coupled because of the difficult relationship between the membership, data sharing, data caching, and messaging. Such a coupling is a problem for both incremental design and composability. OTA began as a simple tracking algorithm and only later incorporated aspects of geographic routing and localization, each time involving a redesign and new decomposition. Once the new components were incorporated, they could not easily be switched with other localization or routing algorithms because they were so tightly integrated with the rest of the application. Section 5 shows how Hood can help address this problem by defining a different relationship between the neighborhood mechanisms.

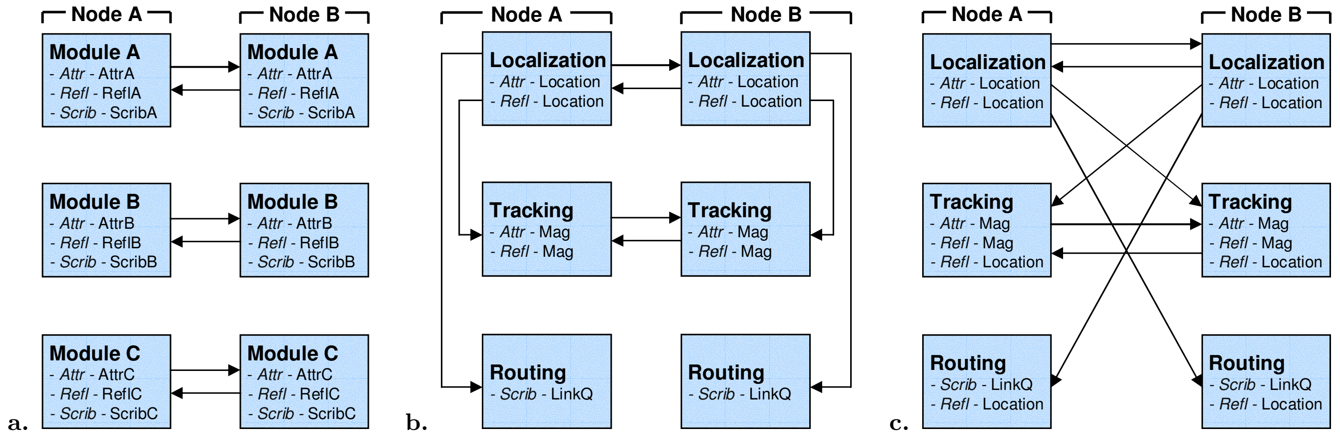

Figure 8: a) It is easiest to isolate data sharing and data caching together within the module that implements the message protocol. b) This can cause coupling of components due to shared membership constraints on the data caches. In this case, Tracking and Routing impose membership constraints on the location cache. c) Replicating the location cache within Tracking and Routing solves the membership coupling, but introduces coupling due to messaging.

5. OBJECT TRACKING REVISITED

The process of designing OTA with Hood as a first-class primitive is different than that of designing with its component pieces of messaging, caching, and neighbor lists. Instead of decomposing the algorithm into message protocols and data structures as we saw in Section 4, the developer defines which data is to be shared and the membership criteria for each neighborhood.

In OTA, two neighborhoods need to be formed, one for tracking and one for routing. The tracking neighborhood needs to reflect both the magnetometer reading and location of each node that is within some geographic radius. The routing neighborhood needs to reflect the location of each node that is closer to the camera than the node itself and has two scribbles to hold link-quality estimators for each node. These neighborhoods, attributes, and mirrors can be created with the following lines of code:

```
generate attribute MagAttr from int;
generate attribute LocationAttr from location_t;

generate neighborhood MagHood {
  wire filter GeographicRadius;
  set max_neighbors to 3;
  reflection MagRefl from MagAttr;
  reflection LocationRefl from LocationAttr;
}

generate neighborhood RoutingHood {
  wire filter ClosestToDestination;
  set max_neighbors to 8;
  reflection LocationRefl from LocationAttr;
  scribble RxLinkQualityScrib from int;
  scribble TxLinkQualityScrib from int;
}
```

The first two attribute commands establish `MagAttr` as an integer and `LocationAttr` as a structure `location_t` defined in a header file elsewhere. When these attributes are set, they are cached and by default pushed to co-neighbors.

The next block generates code for the `MagHood` neighborhood. The block sets the neighborhood filter to be the `GeographicRadius` module and sets the maximum number of neighbors to three. It further defines two reflections in the neighborhood. The `MagRefl` reflection caches neighbor updates of `MagAttr`. The `LocationRefl` reflection caches neighbor updates of `LocationAttr`.

The last block generates code for the `RoutingHood` neighborhood. The block sets the neighborhood filter to be the `ClosestToDestination` module and set the maximum number of neighbors to eight. It futher defines a reflection and two scribbles. The `LocationRefl` reflection caches neighbor updates of `LocationAttr`. The `RxLinkQualityScrib` and `TxLinkQualityScrib` scribbles establish one integer each in the cache for locally derived estimates of the receive and transmit link quality of a neighbor.

The resulting Hood infrastructure for this application is depicted in Figure 10. The two attributes are depicted in the center and associate with the two neighborhoods that span to either side. Only the mirrors of the `MagHood` reflect the magnetometer attribute, but all mirrors reflect the location. If a node were in both the `MagHood` and the `RoutingHood`, its location could be accessed through its mirror in either neighborhood. Other mirrors in the background represent the fact that many more neighborhoods and mirrors could be created by other sub-systems or services that are not considered here, such as the MAC layer or time synchronization service.

This Hood infrastructure supports the overall system architecture shown in Figure 9, where the data cache for all attributes, reflections and scribbles are separate components as are the neighbor lists. This architecture avoids the coupling problems in Figure 8 by separating data sharing from data caching, joining data caching with membership, and providing a data messaging layer to both marshal and dispatch data, and allowing the data-sharing components to communicate with the data-caching/membership components. The algorithms for each system component are also simplified because data-sharing takes place automatically, and neighbor selection is relegated to a separate module.

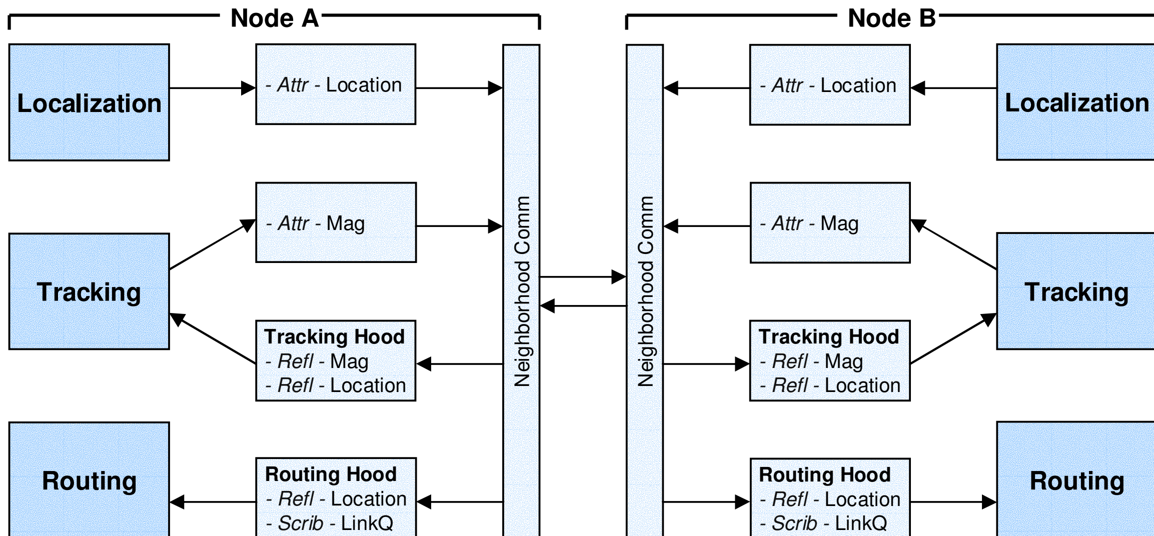

Figure 9: An OTA neighborhood architecture designed using Hood as the fundamental programming primitive. This architecture decouples modules that own and share data from those that need to cache and view that data. This allows modules to express membership criteria independent of the owner modules and other viewer modules. Data is marshalled and dispatched through Neighborhood Comm, avoiding inter-component coupling due to messaging protocols.

- The **localization** component sets the node's location attribute each time it learns that its location has changed.
- The **tracking** component monitors the magnetometer and writes the magnetometer attribute when it significantly changes. If the node is the leader, it estimates the object position using the locally cached magnetometer values and locations of neighbors. It sends an estimation message through the routing component for delivery to the camera node.
- The **routing** component uses geographic routing. Whenever it receives an estimation message, either from the tracking component or from another node, it decides which node is closest to the destination by searching through the cached locations of its routing neighbors.

6. EVALUATION

To compare the algorithmic complexity between the implementations, the tracking modules of both the original and the Hood OTA implementations were analyzed as a state machine. In original OTA it has 6 explicit binary state variables and dozens of states and transitions. Most of these states involve whether or not a data-sharing message is sending, whether or not the magnetometer is being read, the state of the neighbor table, whether the node is performing leader election and whether a tracking message is being sent to the camera. Most transitions involve receiving protocol messages, timer events, magnetometer reading events, and membership changes. In Hood OTA, the same component has only 1 explicit state variable, five states and seven transitions. The one state variable indicates whether the message to the camera is being sent or not. The five states differentiate between collecting data, performing leader election, and sending the tracking estimate to the camera. This reduction in state machine complexity is largely due to the fact that the application does not need to manage the data sharing protocols or the data cache. This decrease in the amount of internal state is significant because stateful behavior must be carefully checked and is often difficult to debug.

The tracking module in original OTA has 212 lines of code while it has 134 lines of code in the Hood implementation, a 33% reduction. The code that was moved from the application module to the filter module was not included as removed code. Almost none of the code savings pertain to membership. About 50% of the savings involve defining, initializing and maintaining state about the neighbor table data structures, and another 50% is saved in message protocol overhead. Of course, *reducing* lines of code in itself is not a measure of better code quality; this statistic is perhaps best read to say that 33% of the code was *replaced* with pre-verified libraries and the Hood API. This is especially important since maintaining data structures and managing messaging protocols are some of the more error-prone parts of an algorithm.

Finally, we compare the coupling of both OTA implementations. In the original OTA, there is tight coupling from the localization module to the tracking and routing modules. Both modules depend on the localization data cache to maintain the location for nodes of interest, and there is no guarantee that locations will be available when the modules need them. Directly improving that cache behavior for the benefit of tracking and routing results in dependencies that make the localization module non-composable in generic applications.

In the OTA built on Hood, the owners of data, such as the localization module, use the Hood library and interfaces to decouple from the viewers of data, such as the tracking and routing modules. Here, location updates are cached

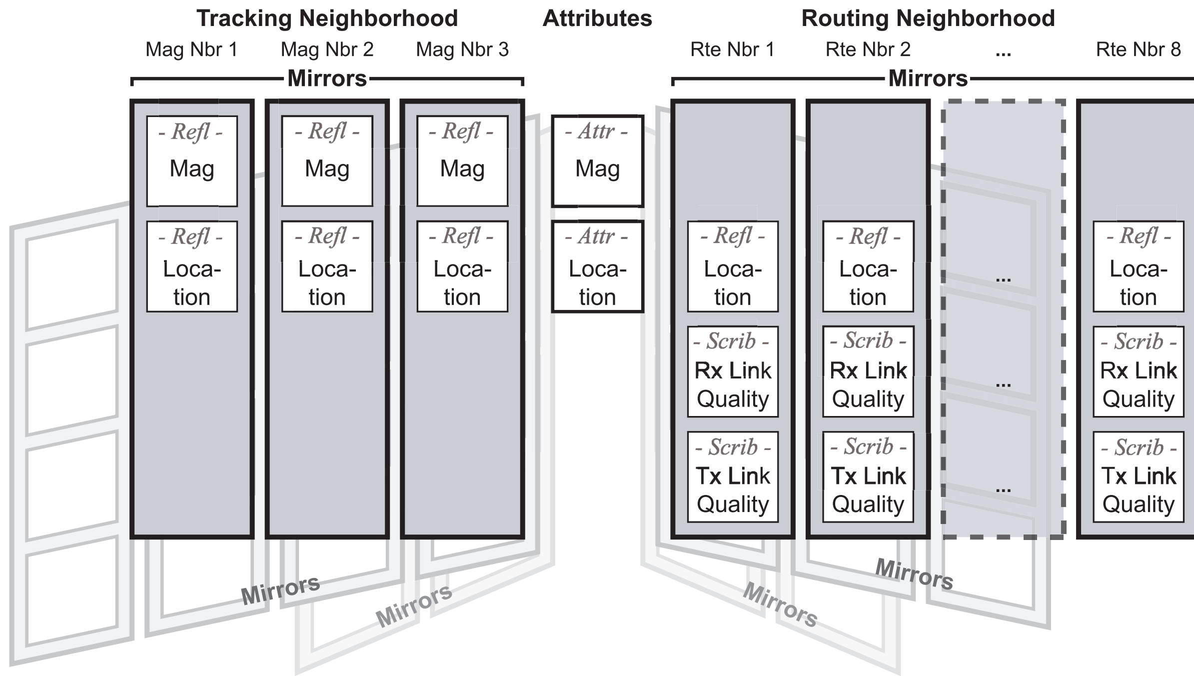

Figure 10: This image represents the interface that each module in OTA has to the neighborhoods, attributes and mirrors created by Hood. The bare white boxes down the middle are the node's own attributes. The larger grey boxes are mirrors and the white boxes inside them represent reflections and scribbles. Notice that `Mag` is reflection only over the `TrackingHood` while `Location` is reflected over both neighborhoods. The grayed-out mirrors represent other neighborhoods that might be created by other services on the system, such as localization or time synchronization.

separately in each neighborhood, assuring that relevant locations are available to each module when it needs them. Furthermore, constraints among mirrors are expressed in a single filter per neighborhood, allowing for high level modules to be reused between applications, for new high level modules to transparently replace old ones without having to express application-specific constraints, and for filters determining the character of a neighborhood to be developed independent of those high level modules. Because of this, the resulting design is more maintainable, composable, and reusable.

7. VALUE TO EXISTING APPLICATIONS

The last two sections have shown that Hood has facilitated OTA both in the design process and in the final code quality, but it remains to be shown that it might benefit any other sensor network applications. To this end, Table 2 lists sixteen existing sensor network systems and services developed at seven different institutions across the country and shows that they all use neighborhood concepts in some form. Hood can capture the essence of these neighborhood usages insofar as they can be decomposed into its constituent components, as shown in the table. Furthermore, many of the implementations are immature enough to warrant the claim that there is still a barrier of entry to using neighborhoods at all. These applications have all been made publicly available either in the TinyOS repository or on the web.

A brief analysis shows the approximate number of neighbor lists, shared attributes, reflected attributes, and local scribbles used in each application. Only those neighborhood instances specific to an application itself were included in the totals. For example, while high level applications like *Object Tracking* seem to use neighborhoods only once or twice, they might use *Location Node, Mutation Routing, Time Sync* and *S-mac* as underlying services, resulting in total usage of Hood concepts indicated in Table 1.

All neighborhoods discovered were one-hop neighborhoods and most if not all of the neighborhoods implicitly use the broadcast/filtering approach adopted by Hood. This is good evidence that Hood captures the common neighborhoods in sensor network applications. There are two types of neighborhoods observed, those with more scribbles than reflections and those with more reflections than scribbles. The first type is more common in MAC implementations and some routing algorithms that try to perform silent link-quality estimation. The second type is more common in localization systems, time synchronization systems, and application-specific distributed algorithms that actually have application state and data that needs to be shared.

Many of the existing neighborhood implementations support the claim that, even with all the building blocks of neighborhoods available, a suitable implementation is difficult to achieve. Most implementations maintain the neighbor list as a member variable array within a component. As shown in Section 4 this can cause a coupling problem common in young neighborhood implementations when different components begin to share neighborhood information. The implementations of data caches range from having multiple

Application	Neighbor Lists	Attributes	Reflections	Scribbles
Object Tracking	6	16	17	10

Table 1: OTA Hood usage Usage of hood concepts in OTA if usages of all subsystems were counted.

Application	Neighbor Lists	Attributes	Reflections	Scribbles
Blast	1	3	3	6
Calamari	2	5	5	2
EnviroTrack	2	3	3	2
Geographic Routing	1	0	1	1
GSK	1	0	0	2
Location Node	2	8	8	4
Mutation Routing	1	4	4	3
Object Tracking	1	1	2	1
Prime	1	0	0	1
PG Routing	2	3	3	7
S-mac	2	3	3	5
Social Net	1	0	0	1
Surge	1	2	2	4
TimeSync	1	1	1	1
TSync	1	3	3	1
TinyDB	1	2	2	1
Tiny Diffusion	2	1	1	7

Table 2: Existing Applications publicly available for TinyOS and their approximate usage of Hood concepts.

attributes all stored in one large `char*` array, to being contained each in separate (type-safe) arrays, to more mature implementations that have neighbor *tables*, which are arrays of structures. Only two applications separate neighbor tables as a separate software component that could be independently verified and reused. None of the implementations besides Hood dissociate data sharing from data caching to allow a single attribute to be shared over multiple membership lists, nor do any include messaging machinery to facilitate automatic data sharing.

A clean, well-defined implementation of a neighborhood abstraction is likely to promote the use of more neighborhoods and reflections. The four systems and services that were implemented with Hood, *Calamari*, *Geographic Routing*, *Mutation Routing*, and *Object Tracking* have an unusually high number of uses of neighborhood compared to other applications in their corresponding categories. This is partly due to the fact that Hood inspired their architectures as well as the lower effort required to maintain lists of neighbors and state about them.

7.1 A Distance-Vector Implementation

To more concretely show that Hood can capture many sophisticated distributed algorithms, this section examines how one of the applications described above was implemented with Hood.

Calamari is a self-localization system for ad-hoc sensor networks [18] that discovers the positions of the nodes in a network relative to three or more anchor nodes. There are two parts to Calamari: ranging and localization. The ranging system gives each node the distance to its neighboring nodes. The localization system uses these ranging estimates with the *DV-distance* algorithm [12] to estimate the node's location. In this section, we describe how both of these components exploit properties of neighborhood.

The *DV-distance* algorithm requires each node to trilaterate against at least three anchor to estimate its position. If a node is not within ranging distance of an anchor node, it can estimate the distance to be the shortest path distance to that anchor. Each anchor initiates a flood where nodes estimate, share, and revise their distances estimates to the anchor nodes. In this way, the anchor node locations propagate through the entire network, simultaneously building shortest path distance estimates in a distance-vector manner.

Importantly, this long-distance information sharing and recursive algorithm are done with only local neighborhoods. When designing this algorithm with Hood, the neighborhoods and shared variables must be defined. Each node needs a `RangingHood` for the ranging estimates of all reporting nodes and a `ShortestPathHood` for its neighbors' shortest path estimates and the corresponding anchor locations. Finally, each node needs a set of local `ShortestPath` attributes for its own shortest paths to each anchor.

Given this infrastructure, the algorithm is easy to understand. Ranging estimates are scribbled into the `RangingHood` whenever they are received. Because shortest path estimates are automatically shared, the node must simply watch for changes in the shortest path reflections and the ranging scribbles. If a change is detected, the local shortest path attribute is set to the new value. These values are automatically shared and are available anytime for computing a new location estimate.

8. WORK IN PROGRESS

The neighborhood abstraction is optimized for lossy datasharing over a one-hop neighborhoods. It is not, however, constrained to this domain and can be extended to multi-hop neighborhoods or to bi-directional, reliable neighborhoods.

This section reviews two natural extensions to the standard abstraction that are currently being developed and the issues they introduce.

8.1 Multi-hop Neighborhoods

A one-hop broadcast mechanism provides an implicit neighborhood, those nodes with which there is radio connectivity, and the membership filters are simply restricting this number to a useful set. To generalize this idea, *any* broadcast mechanism defines an implicit neighborhood. For example, two-hop neighborhoods could be supported by two-hop broadcasts or geographic neighborhoods could be supported by geographically-directed broadcast.

This extension is not quite so simple, however. Recall that an attribute is always pushed to *co-neighbors* and pulled from *neighbors*. This means that a "two-hop to the East" neighborhood would need to be supported by a "two-hop to the West" broadcast mechanism. Furthermore, an attribute *pull* in multi-hop neighborhoods must either be supported by full-fledged routing algorithms or must use eg. a "two-hop to the East" broadcast mechanism, which is wasteful.

To support multi-hop neighborhoods, the NeighborhoodComm component allows one to override the communication protocols necessary to send messages to all potential co-neighbors (*push*), each neighbor (*pull*), or a specific co-neighbor (*pull response*). This allows one to extend the neighborhood abstraction to multi-hop neighborhoods by changing only a single software component. Notably, however, no existing application has yet required anything more than a one-hop broadcast.

8.2 Bi-directional, Reliable Neighborhoods

It may be desirable to have neighborhoods in which the radio links to all neighbors are bi-directional and/or reliable. This is especially important for ad-hoc routing protocols, which almost always maintain state about neighbors and often care only about bi-directional or reliable neighbors.

Choosing bi-directional or reliable neighbors can be achieved by either pinging each neighbor, perhaps periodically, or requesting an acknowledgment after each routing message. The ratio of acknowledged messages can be held in a *scribble* associated with each node.

Bi-directional and reliable neighborhoods require a single caveat. Even though they do not necessarily require a multi-attribute filter, the scribble for each node cannot be thrown away without allowing the possibility of *thrashing*, where a node is ejected from the neighborhood and is soon reconsidered for membership. Furthermore, the potential neighbor cannot provide this information in its membership messages. Therefore, nodes maintaining this type of neighborhood *must* be robust to neighbor thrashing, possibly by having enough memory to hold information about all potential neighbors, or through some other technique.

9. RELATED WORK

Hood is something between a process group abstraction and reflective memory. The *process group* approach [1] has a long history in the distributed computing domain, where *groups* are a globally consistent list of member processes. Process groups have very strict semantics, however, which break down in the presence of network partitions. A series of developments in *partition-aware* group management lead to work by Briesemeister and Hommel [2], who introduced the idea of a Neighborhood Service with which they build local *views* of group membership. Neighborhood Service resembles Hood in the sense that it allows membership asymmetry. However, in Hood a neighbor list is not a view of global group membership; there are no global group concepts associated with Hood. Furthermore, data sharing in Hood is fundamental to discovery; membership is defined by nodes that pass a general filter, not simply active neighbors within radio connectivity or, as in more recent work, those meeting location requirements [14].

Reflective Memory (RM) systems are a form of shared memory popular for parallel systems [10, 13], in which each system's memory is reflected into mirrors in other systems memory. All writes to local memory are always pushed into all mirrors. RM can support multiple-reader/multiple-writer protocols and enforce a wide variety of cache consistency mechanisms, typically employing specialized hardware to do so. Hood reflection mechanisms are something of a degenerate case of RM, where the consistency model is extremely simplified by moving to a multiple-reader/single-writer model. Furthermore, Hood reflection is not a push-only model and does not enforce any cache coherence requirements. Instead, it exposes both a push and pull interface and chooses to employ only a best-effort coherence strategy so that access to data would never be restricted.

In this sense, Hood is related to worm and epidemic algorithms in their concept of eventual consistency [16, 11]. Such algorithms provide that, given enough time, all nodes will eventually share a consistent view of certain data even in the face of network loss or temporary partitions and without the cost of stricter consistency guarantees. While a similar philosophy is taken by Hood, its values are neither anonymous nor globally shared. Each value in Hood is inherently identified with a single node, its origin, and a limited scope, the node's co-neighbors.

Hood is perhaps most similar to the *Home Page* model of *pFrags* proposed by Butera [3] for paintable computers. pFrags are process fragments that move from node to node trying to achieve some goal. Each pFrag can write to a local Home Page which it shares with other local pFrags, and this Home Page is mirrored to some degree on neighboring nodes. Besides the fact that pFrags endorse a distributed computing model while this paper endorses a SPMD parallel programming model, there are two main differences between Home Pages and Hood. First, Hood only allows nodes to write their own attributes, which live in a global namespace so that neighboring nodes can reject attributes they don't recognize. Home Pages, on the other hand, can store any type of data. Second, Hood allows the *remote* node to decide which nodes and which data should be mirrored, whereas Home Pages require each pFrag to decide which of its own data will be shared with *all* other pFrags.

In this sense, Hood is also similar to Blackboard systems [5], which are popular in AI, and tuple spaces like Linda [9], which have become popular in distributed computing. In both of these programming models, a process adds data to a globally accessible repository and other processes *choose* which data to read. Similar to Hood, the producer of the data does not know who the consumers are. In Hood, however, the data is not anonymous but is inherently identified with a single node, and all consumers *do* know the identity of the producer. Furthermore, data remains on a blackboard or in a tuple space independently of the originating process.

In Hood, in contrast, data changes with the state of the originating processes; it more closely resembles shared process state than shared data.

Most recently, Abstract Regions uses a neighborhood abstraction based on that originally proposed by Hood [17]. This abstraction defines the relationship between discovery, data sharing and reductions and has been used to implement several sophisticated distributed algorithms.

10. SUMMARY AND CONCLUSIONS

While some distributed algorithms are oriented around messaging protocols, many others are oriented around neighborhoods and data sharing. The only programming primitives available for sensor networks today, however, are for supporting protocols. Applications that rely on neighborhood-oriented algorithms should not have to reduce those algorithms to protocols and data caches, but should be able to design them directly on a neighborhood abstraction. This paper proposes such an abstraction and shows that it captures the essence of the neighborhood concepts needed by many existing applications.

Neighborhood concepts are being used in many of these applications today, even explicitly so; Table 2 was partially created by searching for the characters *"neighb"* in the code repository. This is essential to supporting the claim that a clean, well-defined neighborhood abstraction is needed. All known existing neighborhood implementations are one-hop neighborhoods and can all be decomposed into concepts similar to those found in Hood: neighbor lists, filters, mirrors, reflections, and scribbles. This supports the claim that Hood can be an answer to the aforementioned need for a neighborhood abstraction.

The primary benefit of Hood over existing neighborhood implementations is that it clearly defines the relationship between several concepts fundamental to neighborhoods, *membership*, *data sharing*, *data caching*, and *messaging*. All other implementations fail 1) to dissociate data sharing from data caching and 2) to integrate neighbor lists and caching with messaging. In Hood, the first is achieved by using *mirrors*, which allow attributes to be reflected over multiple different neighbor lists without requiring coordination between the different lists. The second is achieved by using *filters*, when attributes are pushed, neighbor lists and caches are built up from them, without intervention from or coordination with application logic. This unified abstraction in Hood has been shown through a case study to benefit the design, implementation, and modification of applications such as OTA that already rely on neighborhood concepts.

Acknowledgements

This work is funded in part by the National Defense Science and Engineering Graduate Fellowship, the DARPA NEST contract F33615-01-C-1895, and Intel Research. Special thanks to Matt Welsh for many thoughtful discussions and insight into the nature of the Hood abstraction.

11. REFERENCES

[1] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, December 1993.

[2] Linda Briesemeister and Günter Hommel. Localized Group Membership Service for Ad Hoc Networks. In *International Workshop on Ad Hoc Networking (IWAHN)*, pages 94–100, AUG 2002.

[3] Bill Butera. *Programming a Paintable Computer*. PhD thesis, MIT, February 2002.

[4] D. Chaiken, J. Kubiatowics, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, 1991.

[5] D. D. Corkill. Blackboard Systems. *AI Expert*, pages 40–47, 1991.

[6] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *International Conference on Mobile Computing and Networks (MobiCOM '99)*, Seattle, Washington, August 1999.

[7] Armando Fox and Eric A. Brewer. Harvest, Yield and Scalable Tolerant Systems. In *Workshop on Hot Topics in Operating Systems*, pages 174–178, 1999.

[8] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[9] D. Gelertner. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[10] M. Jovanovic and V. Milutinovic. An Overview of Reflective Memory. *IEEE Concurrency*, 7(2):56–64, 1999.

[11] Phil Levis and David Culler. Mate: a Virtual Machine for Tiny Networked Sensors. In *ASPLOS*, October 2002.

[12] Dragos Niculescu and Badri Nath. Ad Hoc Positioning System (APS). In *GLOBECOM (1)*, pages 2926–2931, 2001.

[13] Sanjay Raina. Virtual Shared Memory: A Survey of Techniques and Systems. Technical Report CSTR-92-36, University of Bristol, 1, 1992.

[14] G.-C. Roman, Q. Huang, and A. Hazemi. Consistent Group Membership in Ad Hoc Networks. In *23rd International Conference in Software Engineering (ISCE)*, Toronto, Canada, May 2001.

[15] Chia Shen and Ichiro Mizunuma. RT-CRM: Real-Time Channel-Based Reflective Memory. *IEEE Transactions on Computers*, 49(11):1202–1214, 2000.

[16] J. F. Shoch and J. A. Hupp. The "Worm" programs - Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, 1982.

[17] Matt Welsh and Geoff Mainland. Programming Sensor Networks Using Abstract Regions. In *The First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[18] Kamin Whitehouse. The Design of Calamari: an Ad-hoc Localization System for Sensor Networks. Master's thesis, University of California at Berkeley, 2002.