Macroprogramming Heterogeneous Sensor Networks Using COSMOS

Asad Awan Suresh Jagannathan Ananth Grama
Department of Computer Science, Purdue University, West Lafayette, IN 47906
awan,suresh,ayg@cs.purdue.edu

Abstract

In this paper, we present COSMOS, a novel architecture for macroprogramming heterogeneous sensor network systems. Macroprogramming entails aggregate system behavior specification, as opposed to device-specific applications that indirectly express distributed behavior through explicit messaging between nodes. COSMOS is comprised of a macroprogramming language, *mPL*, and an operating system, *mOS*. mPL macroprograms specify distributed system behavior using statically verifiable compositions of reusable user-provided, or system supported functional components. mOS provides component management and a lean execution environment for mPL in heterogeneous resource-constrained sensor networks. COSMOS facilitates composition of complex real-world applications that are robust, scalable and adaptive in dynamic data-driven sensor network environments. The mOS architecture allows runtime application instantiation, with over-the-air reprogramming of the network. An important and novel aspect of COSMOS is the ability to easily extend its component basis library to add rich macroprogramming abstractions to mPL, tailored to domain and resource constraints, without modifications to the OS. A fully functional version of COSMOS is currently in use at the Bowen Labs for Structural Engineering, at Purdue University, for high-fidelity structural dynamics measurements. We present comprehensive experimental evaluation using macro- and micro- benchmarks to demonstrate performance characteristics of COSMOS.

1. Introduction

As the applications of sensor networks mature, there is increasing realization of the complexity associated with programming large numbers of heterogeneous devices, operating in highly dynamic environments. Much of the complexity arises due to programming coordinated activity of nodes in order to yield system behavior of the sensor network as a whole. Even conceptually straightforward tasks, from the view point of unified-system behavior, such as data acquisition, processing and aggregation require non-trivial programming effort, especially when accounting for scalability, failures, and constraints on node and network resources.

Traditional approaches to programming sensor networks involve developing "network-enabled" applications for individual nodes. Such applications encode distributed system behavior via explicit messaging between nodes. Furthermore, application development often requires implementation of low-level system details, or reliance on inflexible or inefficient underlying platforms. Consequently, application development is time- and effort-intensive, error-prone, and has limitations with respect to scalability, heterogeneity, and performance. In contrast, macroprogramming entails direct specification of the behavior of a distributed ensemble. Through suitable abstractions, macroprogramming explicitly supports heterogeneity, optimized performance, and scalability and adaptability w.r.t. to network and load dynamics. Furthermore, behavioral specifications can often be statically verified to ensure robust behavior.

Use of a few low-cost and low-power, yet relatively powerful devices (such as X-Scale based Intel Stargate) can greatly augment sensor network performance [29]. Existing development platforms either focus on the development for a network of mote-scale devices (e.g., [9, 8]) or for networks of higher performance machines in the sensor network (e.g., [13]). The lack of vertical integration across heterogeneous devices in existing platforms and programming models leads to development complexity and duplicated effort. Thus, there is a need for a unified macroprogramming model that capitalizes on benefits due to heterogeneous environments. This also requires an underlying runtime environment that can execute on resource constrained motes, such as Mica2, and allows utilizing capabilities (e.g., multi-threading) of resource rich nodes.

The goal of supporting easy-to-use high-level abstractions that hide system-level details is often at odds with requirements on low overhead and flexibility. This tradeoff is the primary determinant of the design choices associated with a realizable macroprogramming architecture. We believe that the underlying architecture must provide a low overhead execution environment for fundamental macroprogram primitives, while naturally supporting rich extensions based on the system's operational requirements or application domain. Among other design objectives for a macroprogramming architecture, the reuse of software components across applications is an important engineering aspect, specially as sensor systems become more commonplace.

With these motivating objectives, we have developed COSMOS, an architecture for macroprogramming heterogeneous sensor networks. This paper details the COSMOS architecture, which is comprised of a lean operating system, mOS, and an associated programming language, mPL. COSMOS supports developing complex applications, based on reusable components. In the COSMOS architecture, aggregate system behavior is specified in terms of distributed data processing. This specification consists of functional components (FCs) and an interaction assignment (IA), which specifies distributed dataflow through FCs. An FC provides a typed declaration in mPL and a C code corresponding to the function. Node capability constraints associated with each FC allow the pro-

grammer to effectively and safely utilize heterogeneous devices in the network. Furthermore, mPL applications can be reprogrammed over-the-air.

mPL supports the expression of IAs as a fundamental primitive with support for static program verification. Synergistically, mOS provides a low-footprint runtime environment providing dynamic instantiation of FCs and IAs along with other requisite OS subsystems. Building on the simple primitive of expressing behavior through composition of FCs, the COSMOS architecture exports the ability to easily append rich high-level macroprogramming abstractions in mPL, without modification to the OS. This is achieved by implementing new semantics as FCs and integrating them through the compiler infrastructure as contracts [16] in mPL. This extensibility gives COSMOS architecture the power to seamlessly cater to widely varying operational and domain specific requirements. Illustrative examples in mPL, presented in this paper, include abstractions for data presentation operators, dynamic load conditioning, and region based aggregation.

We have a fully functional implementation of COSMOS supporting Mica2 and POSIX platforms. On Mica2 motes, mOS runs directly atop the underlying hardware. On the other hand, on POSIX platforms mOS provides a transparent execution environment for COSMOS applications and utilizes the capabilities of the host OS. In addition, mPL programming model allows seamless integration across platforms. We demonstrate the prowess of COSMOS for sensor network development through comprehensive evaluation using macro- and micro benchmarks. COSMOS is currently in use in experiments measuring structural response of buildings to external stimuli at Bowen Labs for Structural Engineering, in Purdue University.

2. Macroprogramming Architecture

The overall goal of COSMOS is to enable an efficient and seamless macroprogramming environment for sensor networks. In this section, we first discuss the macroprogramming abstraction, scope, and rationale for our design choices. We then present the programming model, which allows application programmers to specify aggregate behavior. Finally, we describe the COSMOS runtime system.

2.1 Design Principles

Macroprograms, in the COSMOS architecture, specify distributed sensor network behavior based on the abstraction of distributed dataflow and processing in the heterogeneous network. Heterogeneous sensor networks consists of nodes with varying sensing, processing and communication capability, where failures are a norm. COSMOS comprises of mPL programming language and mOS operating system. As a basic primitive, mPL allows specifying distributed data processing in terms of dataflow through *functional components* (FCs). FCs implement an isolated data processing task, for example an averaging function, using C. mOS provides the underlying runtime for COSMOS macroprograms on heterogeneous platforms.

Basic design principles of COSMOS are:

Macroprogram composition and validation. The macroprogram is composed of reusable components with typed interfaces. The resulting composition is statically verifiable.

Dataflow between FCs. Facilitating dataflow, both local and over the network, between FCs is transparently supported as a runtime primitive. Dataflow is asynchronous. As in most dataflow systems, execution of a component is triggered in response to asynchronous data being available on its inputs.

Transparent abstractions and low-level control. High level abstractions and semantics in mPL language are simply implemented as compositions of FCs. Through the implementation of the FC, low-level details of these abstractions can be controlled. However, these details are transparent to the application. For example, while dataflow over the network is transparent to applications, the network protocol is in fact implemented as an FC that is spliced into the application at compile-time. Other examples of high level abstractions are region communication, load conditioning, and handling failures. A key benefit of this approach is that the underlying mOS remains unaltered when abstractions are added.

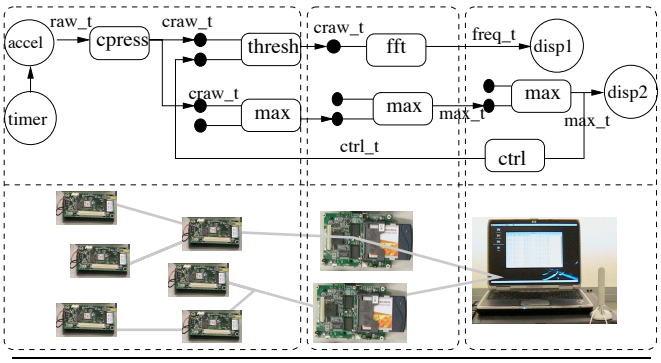

Figure 1. A macroprogram and its instantiation. This program displays max acceleration values for a building and evaluates frequency response of the building, and displays the resulting spectrogram if the acceleration is above a specified threshold. A controller (ctrl) feedbacks the threshold value, based on aggregate data from the network. This conserves system resources untill an interesting event triggers the need for a high fidelity view.

Heterogeneity. mPL syntax allows effectively utilizing heterogeneous nodes in the network based on their capabilities (e.g., if a node has a sensor, or has large memory). This provides a unified programming model that is aware of hardware capabilities. In congruence, mOS is also capability-aware and provides a platform independent runtime for the applications.

Reprogramming. Applications (given as dataflow graphs) can be loaded at runtime by mOS. This allows low-overhead over-the-air reprogramming of the sensor network.

2.2 The *mPL* Programming Model

An mPL macroprogram provides a distributed behavioral specification, and constraints relating to the instantiation of this specification on physical nodes in the network. The specification consists of functional components (FCs) and an *interaction assignment* (IA), which is a directed (possibly cyclic) graph that specifies dataflow through FC instances. Each FC has typed data inputs and outputs, and represents a specific data processing function. FCs are interconnected via logically asynchronous data channels (this allows cyclic graphs). An output of an FC may be connected to the input of another FC, in an IA, only if their types match. Thus, an IA can be statically type-checked. In addition to dataflow, an IA also specifies data producers and consumers, which correspond to source and sink *device ports*, respectively. Device ports are logical and may not always correspond to hardware devices. Figure 1 illustrates an mPL macroprogram used for structural monitoring. We will use this as a running example in this section. Note that each component in this program is instantiated on every node that fulfills the component's capability constraint (depicted in distinct dashed-line boxes). Distributed (dynamic) dataflow is established by the system.

2.3 Functional Components

Functional Components are composed of a typed interface definition and a handler written in C, corresponding to its function. Following are the key features of FCs.

Isolation and concurrency. Logically, an FC is an elementary unit of execution. It is isolated from the state of the system and other FCs in an IA. An executing FC has access to stack variables and a private state memory allocated on the heap at instantiation (i.e., no access to system global variables). This memory model and the isolation properties of an FC naturally allow safe concurrent execution of FCs. As FCs are isolated from the OS, they are insulated from the side-effects of preemption due OS events such as interrupt handling by mOS running on mote-scale devices.

Functional components are non-blocking tasks. Asynchronous data to FCs is received through its inputs interface. If data is available on its inputs an FC is scheduled for execution. This allows a very low-footprint scheduler in mOS that does not need to maintain blocked task lists and thread stacks or provide context switching, which consume precious memory (e.g., $4KB$ on Mica2) and CPU resources. Similar, design patterns are used by most sensor operating systems, e.g., TinyOS [9] and SOS [8]. Additionally, in COSMOS, the architecture of mOS capitalizes on the isolation properties of FCs to execute components as independent threads on resource rich nodes (relying on the underlying host OS). Thus, synergistically, FC and mOS design imposes low-overhead on resource constrained nodes and achieves performance scaling on resource rich nodes, while maintaining a clean abstraction—FC as an elementary unit of functionality and execution—for the developer.

Memory model. At runtime, an FC cannot dynamically request memory. This memory model has two benefits. First, it eliminates memory related failure modes in FC implementations. Second, it simplifies, and hence reduces the overhead of the underlying memory management subsystem. Note that the memory for the data arriving at the FC inputs is dynamically managed by the underlying mOS and is separate from the state memory (cf. § 3.1). Thus, restriction against dynamic memory allocation does not effect FC scalability in face of dynamic load (dataflow) variations. This memory model arises from our experiences in designing FCs — relying only on state memory without worrying about memory failure modes greatly simplifies FC implementation and, in general, increases robustness. If FC implementations were auto-generated from statically verifiable models the restriction against dynamic allocation can be relaxed (cf. § 2.5).

Reusability. FCs are reusable components since they only specify functionality and typed inputs and outputs. Thus, an FC that computes a Fast Fourier Transform, for example, can be plugged into any application as long the type requirements of its inputs and outputs are consistent, and network has nodes that satisfy the capability requirement of the FC. Therefore, FCs provide a robust abstraction for implementing reusable data transformation and aggregation functions that can be applied on data streams.

Extensions. Beyond FCs, with the basic properties described above, COSMOS supports two more types of FCs with extended features. These are service FCs (SFCs) and language FCs (LFCs). SFCs implement system services (e.g., network service) and exist independent of applications. SFCs are linked into the dataflow of applications requiring them at application instantiation time. LFCs manifest themselves as high-level programming abstractions in mPL and are not directly visible to the macroprogram developers. The developer simply uses the high-level programming abstractions provided in mPL, while the compiler transparently selects and inserts requisite LFCs in the application IA during macroprogram static analysis. Further, details on LFCs and SFCs are deferred to Sections 2.5 and 4.3, respectively.

```
%fft : {
   mcap= MCAP_FAST_CPU,
   fcid=FCID_FFT,
   in[craw_t ],
   out[freq_t]
};
```

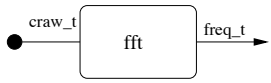

Figure 2. Description of the fft FC meta-information and its graphical representation.

2.3.1 FC Specification and Implementation

The syntactic definition of an FC consists of two parts. The first part contains the FC declaration, specified at design time. The second part is an implementation expressed in a C subset. For a macroprogram developer, only the FC declaration is important. A repository of implementations is assumed to be available as a library. Furthermore, new FCs are straightforward to design and implement. A description of the meta-information (i.e., declaration) of `fft` FC is provided in Figure 2. This FC can be executed only on machines with fast CPUs, has a unique ID (`FCID_FFT`), an inputs of type `craw_t` (compressed raw), and an output of type `freq_t` (frequency domain data). The C implementation of this FC is given in Figure 3. Briefly, the C implementation simply involves writing a handler (`fft_fc()`). This handler receives three parameters from mOS, when it is invoked. The `ind` parameter gives the index of the input of the FC on which data was received, and the `pbqn` pointer points to the input *data block*, generated by the preceding FC in the IA, wrapped in a generic data structure. In this example it contains data of type `craw_t`. The `craw_t` data (that represents a vector of sensor readings, each of which is of type `uint8_t`) is fed to a public domain FFT implementation [4] (modified to remove global variables and malloc calls). The resulting frequency domain data along with other fields of `freq_t` type are transfered as single block to the 0^{th} index output using `out_data()` non-blocking system call (cf. § 3.4). This call enqueues data to the inputs of succeeding FCs in the IA and schedules them if necessary. Further discussion on the features of the handlers and facilities provided to them is deferred to Section 3.4. This example demonstrates the relative ease with which complex data processing functions can be integrated into COSMOS. It also illustrates the modular nature of FCs, that enables them to be plugged into any application conditioned on typed-interface and machine capability definitions.

2.4 *mPL* Language

A macroprogram in mPL is composed of the following fundamental types of expressions.

Enumerations. Enumerations associate integer values with identifiers. These are typically used for providing globally unique IDs. Enumerations follow C `enum` expressions and are automatically imported from COSMOS C header files by the compiler.

Declarations. These expressions allow declarations of devices and FCs. These are also directly imported from C header files, where they are specified in comments using formal syntax.

Instances. Much like declaring objects as instances of classes, these expressions allow creation of logical instances of FCs and devices, and providing instantiation parameters. However, note that, for each logical instance a physical instance is created, at runtime, on each node that fulfills the capability requirement for the component.

Capability constraints. FC (or device) instances inherit capability constraints from FC (or device) declarations. These constraints can be made stricter for individual instances using constraint expressions.

IA description. This section comprehensively describes the connections of the inputs and outputs of FC and device instances used

```
static fc_header_t fc_hdr DECLARE_HDR = {
   fcid: FCID_FFT,
   state_sz: sizeof(fft_state_t)
};

cp_return_t
fft_fc(cp_param_t *param, bq_node_t*pbqn, num_conn_t ind)
{
   kiss_fftr_cfg pc; // using kiss fft implementation
   int nfft;
   uint8_t *d;
   fft_state_t *pfs = (fft_state_t*) param->state;

   if (ind == 0) {
     // craw input.
     register int i;
     nfft = GET_DATA_LEN_CRAW(pbqn);
     d = GET_DATA_POINTER_CRAW(pbqn);
     for (i=0; i<nfft; i++) {
        pfs->in[i] = d[i]; // copy uchar to in type.
     }
     pc = kiss_fftr_minit(nfft, 0, pfs->fft_cfg, CFG_SZ);
     if (pc != NULL) {
        kiss_fftr(pc, pfs->in, pfs->out_freq.data);
        pfs->out_freq.id = GET_ID_CRAW(pbqn);
        out_data(param, 0/*out index*/, &(pfs->out_freq),
                 ID_SZ+sizeof(kiss_fft_cpx)*(nfft/2+1));
     }
   }
   return FC_RETURN_OK;
}
```

Figure 3. Implementation of the fft FC in C.

```
// declarations (auto import)
%accel_x   : mcap = MCAP_ACCEL_SENSOR,
             device = ACCEL_X_SENSOR, out[raw_t];
%cpress_fc : { mcap = MCAP_ANY, fcid = FCID_CPRESS,
               in[raw_t], out[craw_t] };
%thresh_fc : { mcap = MCAP_ANY, fcid = FCID_THRESH,
               in[craw_t, ctrl_t], out[craw_t] };
%ctrl_fc   : { mcap = MCAP_ANY, fcid = FCID_CTRL,
               in[max_t], out[ctrl_t] };
%max_fc    : { mcap = MCAP_ANY, fcid = FCID_MAX,
               in[craw_t, max_t], out[max_t]};
%disp      : mcap = MCAP_UNIQUE_SERVER,
             device = DISPLAY, in[ * ];
%fft_fc    : { mcap = MCAP_FAST_CPU, fcid = FCID_FFT,
               in[craw_t], out[freq_t] };

// logical instances
accel_x   : accel(12);
disp      : disp1, disp2;
cpress_fc : cpress;
thresh_fc : thresh(250);
max_fc    : max;
fft_fc    : fft;
ctrl_fc   : ctrl;

// refining capability constraints
@ on_mote = MCAP_ACCEL_SENSOR : thresh, cpress;
@ on_srv  = MCAP_UNIQUE_SERVER : ctrl;

start_ia
timer(30) -> accel;
accel     -> cpress[0];
cpress[0] -> thresh[0], max[0];
thresh[0] -> fft[0];
fft[0]    -> disp1;
max[0]    -> ctrl[0], disp2 | max[1];
ctrl[0]  --> thresh[1];
end_ia
```

Figure 4. mPL code for the macroprogram in Figure 1.

in the macroprogram. The mPL compiler flags type mismatches and illegal linkages.

Contract predicates. These allow utilization of high-level abstractions in the IA description.

Figure 4 presents the mPL code for the macroprogram in Figure 1. Contract predicates used in this program are not shown; discussion on their usage is deferred to Section 2.6. Device and FC declarations are shown for clarity and, in general, are not written in an mPL code file. Note that the constraints for `fft` imply that its instance should also be present on the server (because they have `FAST_CPU`s as well), however, Figure 1 does not show this detail to maintain clarity.

In the program, two logical instances of the display device are declared using instance expression. This corresponds to displaying two separate windows on the server screen. The input for `disp` is of type "`*`", which is a supertype for all types. The accelerometer is declared using a parameter, which is explained in Section 4.2. In the instance expression for `thresh_fc` we note that a parameter is provided. This is interpreted by the `thresh_fc` as the default threshold value to use.

An IA description is a representation of the IA graph. `timer` is a keyword in mPL. Its parameter (which is tunable at runtime) gives the firing interval in milliseconds. An FC instance on left side of an arrow needs to provide the index of its output for which adjacency is being established. Similarly, FC instances on the right provide the consumer input index. Thus, in the example IA, the first output of `ctrl[0]` is to be connected to the second input of `thresh[1]`. Devices do not need these indices since they have only one input or output. Use of "`->`" indicates local or one-to-one network communication (the optimal choice is discerned automatically by the compiler), while "`-->`" indicates one-to-all communication. The use of "`|`" allows "merge and update" data processing. While generating subgraphs for different node types, the mPL compiler checks if the consumers on the left side of "`|`" are present on the same node as the producer. If so, it creates a local dataflow, otherwise it provides instructions to setup a network dataflow to the consumers on the right side of "`|`".

This example illustrates that an mPL program can easily and directly specify distributed system behavior using simple syntax. Composing applications with reusable components allows the mcaroprogrammer to focus on the application specification rather than low-level details or inter-node messaging. Furthermore, high-level abstractions 2.5 can be used to refine and fine-tune the semantics of the applications (some of which the reader might have noticed missing in our example code) and provide rich features such as neighborhood communication and load conditioning.

2.4.1 Application Instantiation

The COSMOS compiler reads the mPL code and generates an annotated directed graph representation. To execute an application, this graph is communicated over the network nodes running mOS. It is assumed that the requisite FCs are available at the nodes. FCs are placed in the program memory of the nodes either when mOS is being flashed or by downloading them over-the-air. The node capability constraint determines if a node receives a particular FC or not. Each node in the network receives a subgraph of the IA, consisting only of FCs that can be instantiated at the node. Dataflow with remaining FCs is transparently handled through the network SFC (cf. § 3.3). Local dataflow (cf. § 3.1) is established by the mOS runtime.

2.5 High Level Abstractions

Language functional components (LFCs) are used to implement semantics for high-level abstractions in mPL. These abstractions are

specified as contract expressions. Benefits of design-by-contract are well-articulated in both the software engineering [16] and programming languages [5] literature. Contracts are applied on dataflow paths in a macroprogram and are transparent to the underlying OS. A contract expression has the following syntax:
identifier property=const[, stream,...][exception:stream,...][out:stream,...]
The identifier and the property are used by the compiler to lookup the LFC(s) that implement the required semantics. These LFCs evaluate the predicate expression as well as provide mechanisms to meet the specified contract. LFCs are automatically spliced into the user program at compile time. In the above contract expression the constant is an instantiation parameter for the LFC(s), while input, exception, and output streams can be optionally provided. For example, an input stream would be the output of a user FC such as `efcee[0]`. Similarly exception and output streams feed to user FCs' inputs. The type required for the streams (including the one on which the contract is applied) depends on the high-level abstraction used. In the COSMOS architecture LFCs are generated automatically from a temporal logic based program. As these LFCs are verifiable they are allowed to use dynamic memory (which is not allowed for user FCs). A complete description of the programming and the performance of these LFCs is beyond the scope of this paper and is the topic of a separate technical report [3]. In this section, we present a few examples of the use of contract based abstractions supported in mPL. This demonstrates the ability of mPL to seamlessly enable high-level abstractions without modification to the runtime.

2.6 Data Presentation

To motivate the use of data presentation abstractions we continue with our example macroprogram of Figure 1. The `fft` component requires a data vector of specified length. This would require the `fft` component to first build a vector of its required length. This task is further complicated by the fact that `fft` receives an arbitrary number of virtual streams (one from each source node). Segregating these streams and generating required length vectors is orthogonal to the function of the `fft` component. Thus, mPL provides the following generic contract based abstractions that can be used to express its connection with `thresh`:

```
thresh[0] (buffer length=NET_LENGTH) ->
          (buffer length=FFT_LENGTH out: fft[0])
```

Use of `NET_LENGTH` prevents small packets (`FFT_LENGTH` is usually too large and not suitable for transmission in a wireless environment). In response to the use of this contract, the compiler simply places `buffer` LFC at appropriate links in the IA graph during IA graph generation. The `buffer` LFC provides semantics, i.e., buffering and binning, to enforce this contract. Related data presentation abstractions include `geographic_align` and `time_align` property based contracts to align multiple virtual streams (in many-to-one communication) through buffering. In our example program `time_align` is used to sort time sequenced data to the second input of the `max` FC.

2.7 Region Communication

Region communication allows all-to-all communication within a given region. Thus, instead of using a single controller for `thresh`, one could use a local controller that makes its decision based on regional data, e.g., based on data from neighbors in 2-hop radius. The following expressions would achieve this:

```
max[0]  -> (region hops=2, local_ctrl[1] out:max2[1])
max2[0] -> local_ctrl[0]
```

Here, we assume that `local_ctrl` has two outputs, one to control threshold and the other to control the fidelity of its source – in this case to control the diameter of the region. This is also an example of developing sensor network applications that trade off fidelity and resource utilization (radius of neighbors affects network overhead). Figure 5 depicts a part of the IA *generated* by the compiler for nodes with capability `MCAP_ACCEL_SENSOR`, for this example.

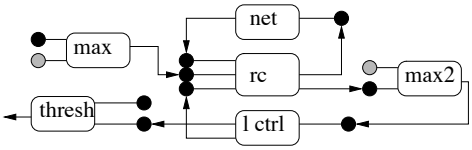

Figure 5. Part of the compiler generated IA for threshold control, using region communication, on sensor nodes.

2.8 Load Conditioning

In a large self-organized distributed system, the number of peers of a given node may not be predictable. Consequently, the input data rate and resource utilization on a node may be arbitrary. Contracts provide system performance control in such environments. Load conditioning LFCs implement simple load-shedding techniques and use the exception feature of contract expressions to allow users to handle contract breaches. Contracts for load conditioning can be based on CPU usage, memory watermark, or network utilization properties.

3. Runtime

In this section we describe the runtime environment, including various features and mechanisms required to support the execution of the macroprogram. We also discuss how the design choices made for the programming model tie to the efficient and robust construction of the underlying environment. The fundamental design pattern in architecting our runtime is to achieve a low-footprint, in terms of CPU and memory, and allow performance scaling on resource rich nodes.

3.1 Dataflow

A macroprogram specifies connections between FC and device port instances using abstract asynchronous data channels. Abstract data channels are realized as data queues, whose nodes encapsulate data. Each device port or an FC instance has an individual queue associated with each of its outputs. During application initialization, these queues are attached to the inputs of the succeeding component as specified by the IA. The two relevant design choices here are – the use of output queuing, and using multiple queues instead of a single multiplexed system wide message queue (which is a simpler alternative, e.g., as used in SOS operating system [8]).

We choose output queuing for our system because it minimizes the memory used by queued data objects. A common case in data processing systems is to extract multiple views of the same data (e.g., finding the average and the maximum of a stream of sensor readings). This requires one-to-many connection of components in the IA. A single output queue shared by all consumers maintains only one copy of the data, thus optimizing a common usage of the system. Of course, the data node is deleted only when all consumers have absorbed the data. However, negative side-effect of having a shared queue requires that the consumers treat inputs as read-only. Given that motes usually have very limited data memory (e.g., Mica2 motes have a 4KB RAM) the memory saving is substantial in practice, and justifies our approach.

The use of individual queues for each output has two key advantages: low CPU overhead, and possibility for concurrency between executing components. Low CPU overhead is a key benefit for resource constrained nodes, while possibility of concurrency allows efficient utilization of resources at resource-rich nodes in a heterogeneous environment.

A single message queue architecture requires source and destination module IDs to be specified in the message. Given that our FCs do not know about the FCs they are connected to, the system would need to include these IDs for every message. This would also require a lookup on the destination FC each time (which also generates more runtime failure modes). On the other hand, only a one-time lookup is required if queues between connected FCs are setup during application initialization. The memory overhead of using multiple queues is compensated by avoiding source and destination component ID fields in each message and instead storing this information only in the queue data structure. Note that a single data object might be destined for multiple FCs and must carry the IDs of each of these FCs. The overhead breaks-even if a few objects are queued. In fact, in high utilization periods, when memory is scarce, we also save memory.

The runtime system also handles network data flow transparently. If the components are not on the same node, output queues connect to a transparent network service of the source nodes. An output queue from the network service of the receiving node to the destination component completes the virtual data channel. The asynchronous semantics ideally suit the network data channel and impose few restrictions on the semantics of the underlying network communication, eventually allowing the use of simple low overhead protocols. COSMOS provides a default network service discussed in Section 3.3.

3.2 Locking and Concurrency Model

COSMOS supports both multi-threaded and non-preemptive environments. On motes, where mOS scheduling is non-preemptive, interrupt handlers can preempt scheduled tasks. To protect globally shared data, the only locking mechanism is to disable interrupts. By reducing the scope of shared data to queues between connected components, we completely eliminate locking on the dataflow path (in a non-preemptive environment). Recall that FCs do not use any global variables. Hence, the only time locks are acquired while executing an application is while scheduling components using the system schedule queue, and during execution of device (including timer) driver routines that share global system data.

On resource rich nodes, where multi-threading is possible, the isolation properties of FCs naturally support parallel execution. However, locks are required between concurrently executing FCs that share a data queue. Our optimization here is to hold the lock for a queue only when enqueuing, dequeuing, or traversing the queue. Thus, while the head object on a queue is read by an executing FC, it does not need to hold the lock for the queue. This allows the FC owning the queue to concurrently enqueue data to the queue. Similarly, as discussed earlier, each queue might have multiple readers. Each of the readers may be reading an object (or different objects) in the queue in parallel. Using a reader bit mask for each object the system knows if an object has already been consumed by an FC. If all readers have consumed the object, it is dequeued and deleted.

3.3 Network Service

The network SFC implements the network communication semantics of macroprograms to enable distributed data flow. During static analysis of mPL program code, the compiler also places the network FC in the generated IA graph if network dataflow is required between the producer and consumer components (i.e., FCs or device components). It is the task of the routing implemented in the network SFC to deliver the data to the destination machine that has consumer components.

We provide a default routing implementation called hierarchical tree routing (HTR). HTR is a simple variation over tree routing in that the nodes near the root are more powerful (in terms of CPU, memory and bandwidth). Tree routing is commonly provided in most sensor network platforms (e.g., [9, 20, 8]). It is scalable, resilient in dynamic environments and is suitable for a wide array of data-driven applications. HTR, in addition, allows easily locating nodes with a particular capability, which is required in our architecture.

HTR provides the following communication semantics. Dataflow between FCs connected over the network can be one-to-all (e.g., `ctrl` (controller) to `thresh` (threshold) FC in Figure 1), one-to-one (e.g., `thresh` to `fft` FC, which yields a many-to-one aggregation at `fft`), or local broadcast (which is used by region communication LFC). By defining group membership based on capability, multicast can be used instead of broadcast. On mote devices communication is best-effort (unreliable), whereas on resource rich nodes TCP is used for data[1] communication. Extensions to these semantics to provide richer (and possibly stricter) abstractions is possible by adding new LFCs transparently. This is exemplified by the region communication LFC that allows neighborhood communication (i.e., all-to-all in bounded region) irrespective of the tree routing.

In general it is difficult to envision a network SFC that would be efficient for all application domains. Thus, while HTR is suitable for data-driven applications, geographical routing may be more suitable for in-network storage based applications (e.g., [21]). A key feature of COSMOS architecture is that the network SFC can be easily modified to adapt to different application domains, without affecting the programming model.

3.4 FC and Runtime Interactions

Runtime parameters. The runtime invokes the FC handler with three parameters (see Figure 3). As described earlier, `ind` gives the input index on which the data is received, while `pbqn` gives the pointer to the input data block wrapped in a generic OS data structure. The `param` argument points to a structure that contains notifications from the operating system, a pointer to the state memory, instance parameters, and (possibly) platform dependent information. Currently, the notifications from the OS include the input queue length. This allows the FC to possibly adept its behavior based on input load.

System calls. mOS supports non-blocking system calls from the FC to the kernel. For example, the `out_data()` function that enqueues data to the output of an FC is a system call. Note that non-blocking system calls suffice in our architecture because all asynchronous data (e.g., data from sensors, network, timers, etc.) is received through the input interface of components. Avoiding blocking calls yields a low-footprint scheduler in mOS, as discussed earlier.

Most system calls used in user FCs pertain to getting information from the OS, such as hardware type, wall-clock time, node id, etc. An exception is the `out_data()` call. Service FCs and language FCs, have a wider range of accessible system calls. For example, network SFC can enqueue data to the network driver, while LFCs can allocate dynamic memory. SFCs and LFCs use the extended system call set to provide low-level control of the underlying hardware and the OS.

FCs are dynamically loadable, hence, a mechanism to enable FCs to access the system call table (which contains pointers to system functions) is required. On motes, this is achieved by placing the table at a known address in the ROM and providing wrappers in a header file to call the actual functions. This technique is adapted from the SOS operating system [8]. In the POSIX environment we simply pass a pointer to the system call table in the `param`

[1] UDP broadcasts are used for routing messages

argument to the handler. Wrappers that transparently invoke system calls based on the underlying platform are provided.

To allow the system to identify the calling FC, an FC is required to pass its system provided parameter (`param`, c.f. Figure 3) in every system call.

Return values. Return values from an FC handler allow it to flexibly control the input queue (on which data arrived causing the invocation of the handler). These "commands" include the following notifications:

1. Current node is no longer needed (`FC_RETURN_OK` as in Figure 3). Recall that a node is deleted only if all its consumers send deletion notification.
2. Transfer the current node without modification to a given output of the FC. If the current FC is the only consumer of the queue a low-overhead pointer transfer is performed otherwise a copy is transfered.
3. Resend the current node to the FC when new data on the current input arrives.
4. Resend the current node to the FC when new data arrives on another input arrives. This allows the FC to synchronize its inputs.

Dynamic memory management of the queues greatly simplify the design of the FCs and enable scalability – they allow the design of the FC to be independent of the data input rate. Data input rate may vary because of the number of nodes in the network or the sampling rate of sensors. Recall that queue length notifications (in the parameters to the handler) can be used to estimate "forward pressure" on the FC so they can adept their behavior if necessary. Alternatively, load-conditioning abstractions can be developed and applied on the dataflow paths.

4. Operating System Architecture

The *mOS* operating system provides a low overhead implementation of the runtime system for COSMOS based macroprogramming. Architecturally, mOS consists of a core kernel, which is logically divided into a platform independent core and hardware specific drivers and routines. The key subsystems of the platform independent core include the scheduler, timer, dynamic memory manager, dynamic component loading and dataflow setup manager, dataflow API, device abstraction layer, and a system information and control API. We have implemented the mOS operating system on Mica2 and POSIX platforms. On motes, mOS is a full-fledged operating system directly running atop the underlying hardware. On resource rich nodes (e.g., in our case, Stargate SBC devices and PCs running Linux), mOS sits atop the POSIX layer, and provides a transparent environment for executing macroprogram applications, which are, by design, platform independent. To execute a macroprogram, on POSIX, the mOS management thread loads FCs (which may be a part of the binary of mOS or compiled as individual loadable libraries), and executes the FCs (waiting for inputs) each in an individual thread. Both on POSIX, and on the motes, mOS is accessible to the dynamically loaded components through a system call pointer table.

4.1 mOS Subsystems

The key subsystems of the platform independent core include the scheduler, timer, dynamic memory manager, application initialization manager, application execution manager, device manager, a network API, and a system information API. Here, we provide a brief overview of a few key subsystems.

The timer subsystem uses a delta list to trigger registered events. On timeout the timer invokes a callback function, or schedules the callback function depending on the usage of the timer API.

The memory manager provides an $O(1)$ free-list implementation to allocate fixed small size blocks, and an $O(n)$ first-fit implementation for arbitrary size memory allocation. If there are no free blocks in the free-list the implementation always falls back to the first-fit implementation. On resource rich nodes, mOS falls back to `malloc()`, if the initially allocated memory chunk is not sufficient.

The network API provides a wrapper function, which is invoked by the drivers when a packet is received. This wrapper invokes the network SFC asynchronously, transparently hiding the underlying driver. Similarly, a single system call to send packets to either of the network interfaces of the node is provided. On Mica2 motes, communication drivers include a driver for the UART and CC1K radio. On POSIX platform an IP driver (that wraps UDP and TCP sockets) and the serial port (UART) driver are provided.

The system information API provides a simple API to access the current node's capabilities, system time, and a unique node ID, among other system information.

4.2 Device Ports

mOS supports an abstraction of device ports. Given a device name, which identifies a device driver, the device subsystem binds a virtual device port to the device driver. This may be associated with a specific hardware device, such as ADC, or a virtual device, e.g., graph plotting application. Each device port has an output data queue, linking it in the IA. It is not possible to insert data in this queue in an interrupt service routine. If this were allowed, access to this queue by consumers would require disabling interrupts. Thus, once a data object is created, a function to enqueue it is scheduled. We allow the macroprogram to specify the number of data samples that can be packed into a single data object before it is enqueued. This provides substantial performance improvement at high sampling rates. In the mPL code this specification is expressed as a parameter to the device port instance declaration. Furthermore, this parameter is tunable at runtime. An example instance declaration specifying the requirement to pack 12 samples per output node is as follows:

```
accel_x   : accel(12);
```

4.3 Supporting Services

Service FCs (SFCs) have access to low-level system functions through an extensive set of system calls that enable them to interact with the system. The key difference between user FCs and SFCs is that service FCs can exist without being a part of an application and are initialized at boot-time. By default, each service has a queue attached to its (0^{th}) input. This queue enables the OS to communicate with the SFCs asynchronously. The above two features allow SFCs to perform low-level management that does not directly involve user data processing.

Alternatively, SFCs can be spliced into user applications when the macroprogram is loaded. The inputs (except 0^{th} input) (and similarly outputs) of a service FC are of the same type. A service can support arbitrary number of inputs and outputs (which is unlike user FCs where the number of FCs are defined at design time). Therefore, only one instance of a service is required on each node.

Usually, SFCs are used to implement complex interaction with sophisticated devices or perform hardware management.

4.4 Application Loading

To instantiate an application both the loadable binary of the FC and a part of the IA corresponding to the current node's capability should be present on the node. Note that upon over-the-air download the binaries of the FC implementation need to be loaded to the program memory. On POSIX based environments, we use the host OS provided dynamic library loading functions (e.g., `dlopen`), and get a handle. On motes, specific drivers are needed to write to the

program ROM at runtime. For Mica2 we adapt this implementation from SOS [8].

The application initialization manager reads the IA and creates data structures for the FCs and device ports and connects them using data queues. Where required, it also connects the FCs (or device ports) to the network service providing a virtual continuation of the IA to the rest of the distributed system. Each FC is represented in the kernel using a *connection point* (CP) data structure. This dynamically allocated structure keeps pointers to the input and output queues, pointer to the handler function of the FC, pointer to the runtime parameter for the FC (i.e., `cp_param_t`) and scheduling information. The initialization manager is also responsible for initializing and managing service FCs.

Once the initialization of the local IA is complete, the application is launched. Starting an application involves enabling data input to the local IA. This entails calling the start function of the device ports and/or marking the output queues of the network service that connect to FCs in the IA as ready.

4.5 Scheduling

Scheduling responsibilities of the mOS operating system involve scheduling FCs that have data ready on their inputs, and to schedule the internal tasks of the operating system. Our scheduler provides light weight, two-level priority scheduling of callback function pointers. mOS scheduler by itself is non-preemptive and dispatch involves invoking the callback function. High priority is used only for system related tasks.

FCs are scheduled on motes as follows – Each time data is inserted into an output queue a system function, `run_cp()`, is scheduled. This function invokes the handler of FC's consumers. Similarly, if the consumer of the queue is a device, a uniform device execution function, `run_dev()`, is scheduled. Timer based execution of device ports or FCs involves scheduling these `run_xxx()` functions on timer interrupts.

In the POSIX environment COSMOS utilizes the underlying host OS scheduler and POSIX threads for concurrency. Each FC has a wrapper that invokes the FC. The wrapper function is executed in a POSIX thread. This function continually performs a conditional wait to block for data. When data is enqueued to an empty queue that feeds an FC, the corresponding FC thread is signaled by calling a POSIX aware `run_cp()` implementation. If the thread is not in a wait state, no signaling needs to be performed. Since enqueue and dequeue operations are atomic (on POSIX platforms we use mutex locks) race conditions are avoided. Similarly, device driver implementations also execute as threads. For example, the IP driver and the UART driver use receive and send threads.

4.6 Implementation

We have implemented the mOS operating system on Mica2 and POSIX platform. The platform independent code consists of approximately $9,000$ lines of C code. The core is implemented to be lean in terms of program size, and CPU / memory usage. We have deployed and are using COSMOS on a real-world three-story building testbed to study structural response to ground motion, in the Bowen Labs [1] at Purdue University.

Since the core of the OS is platform independent it is relatively straightforward to port mOS to new platforms. This involves providing drivers matching the uniform device subsystem of mOS. On Mica2, mOS (including drivers) compiles down to a 25KB binary (the total available ROM size is 128KB), leaving around 100KB for functional components. Out of the 4KB RAM the OS uses around 900B of RAM and has 1748B available in the heap after the OS boots up. Almost 70% of the Mica2 platform dependent code and drivers have been ported from TinyOS [9] and SOS [8] operating systems. These include drivers for the ADC, UART, and routines to access the program memory.

While COSMOS allows user to implement efficient data processing applications, users may want to use external applications in conjunction with COSMOS on the POSIX platform – for example, using a database server application to store results. To achieve this, users only need to develop a driver that communicates with such applications and provide a service FC to control the device. They can then integrate the service FC into the macroprogram IA seamlessly.

5. Experimental Evaluation

COSMOS provides a low-overhead platform for macroprogramming sensor networks. In this section, we present a comprehensive experimental evaluation of various operational aspects. They layout of this section is as follows.

As a macro benchmark, we study the performance of the example program shown in Figure 1. The example application is a real-world structural health monitoring application with high data rate and processing requirements. COSMOS allows a robust implementation of this application in heterogeneous sensor networks. The results demonstrate the advantage of vertical integration across heterogeneous nodes, which is seamlessly enabled by the mPL programming model in congruence with mOS operating system. Next, we demonstrate that COSMOS allows low-overhead over-the-air application reprogramming.

We provide detailed micro-benchmarks to evaluate the footprint of mOS operating system. The footprint of mOS matters the most on resource constrained nodes. Therefore, our micro-benchmarks are focused on Mica2 motes. We evaluate the performance of mOS using benchmarks that stress operating system primitives. We compare mOS with TinyOS [9] and SOS [8], based on CPU utilization. The results show that mOS performs slightly better than SOS. The CPU utilization of TinyOS is lesser than mOS by a fixed absolute difference of 1%, which is low. We show the breakdown of processing costs incurred on the data path on mOS. Finally, we show that complex FC interaction assignments—with several FCs connected in series—have a low footprint on mOS.

5.1 Hardware Setup

We use the following hardware in our macroprogram evaluation benchmarks. Mica2 with MTS310 multi sensor board [2] is used a sensor node. Mica2 has an ATmega128 micro-controller running at 7.37MHz, with a 128KB program ROM, 4KB data RAM, and Chipcon CC1000 radio. MTS310 board supports a wide variety of sensors including a 2-D accelerometer, which is used in our experiments. Stargate [2] is used as a low-power gateway. Stargate has a 400MHz Intel X-Scale (PXA255) processor and provides a 64MB SDRAM and a 32MB Flash. It runs a Linux 2.4.19 kernel. Wireless networking is achieved using AmbiCom Wave2Net 802.11b CF-slot wireless card. To interface the Stargates with the Mica2 radio network we attach a Mica2 mote to the Mica2 slot (which uses UART for communication) on the Stargate. This Mica2 node (network interface mote, or NIM) simply forwards radio packets to the Stargates and messages from the Stargate to radio. Hence, NIMs are transparent to the network and do not perform any sensing. Finally, in our hardware setup an Intel Pentium 4 1.70GHz PC with 512MB RAM is used as a server. The server runs Linux 2.6.17.7 kernel. Wireless networking is provided through a 3COM 802.11a/b/g PCI card. The server can interface to the Mica2 radio network using a MIB510 [2] board connected to the serail port. The MIB board hosts a Mica2 NIM, which acts as a transparent interface to the radio network.

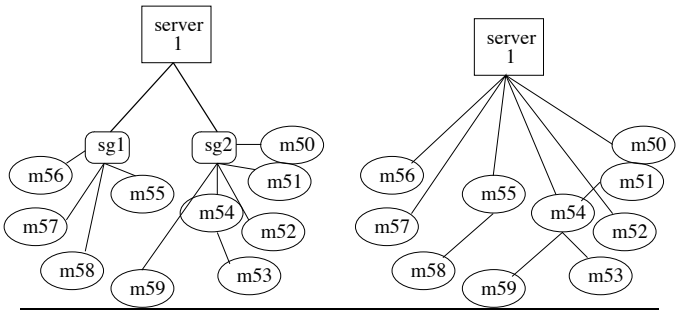

Figure 6. The figure on the left shows the routing tree for a heterogeneous network with two Stargate devices. Figure on the right shows the tree for a network with the server directly communicating with the motes.

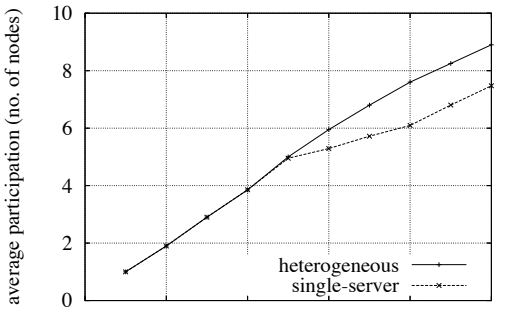

Figure 7. Average participation of nodes in performing *max* data merging over the system. The higher the participation, the higher the quality of the max data recorded.

5.2 Macro Evaluation

We evaluate the macroprogram illustrated in Figure 1. The code for the macroprogram (cf. Figure 4) was changed so that output from the `fft` and the `max` components, at the server, is stored in a file using the `fs` driver of mOS, which replaces the `disp` driver components in the provided code. This example represents a real-world application for structural engineering with high performance and fidelity ($>$ 33Hz sampling) requirements.

5.2.1 Setup and Objectives

The experiment uses 10 Mica2 nodes, 2 Stargates (with NIMs), and a unique server. For this experiment the macroprogram application is "flashed" into the devices (i.e., not programmed over-the-air, which is dealt in Section 5.3). The experiment is run in a large laboratory environment. One of the key features of COSMOS macroprogramming model is that it provides seamless vertical integration over heterogeneous devices. Through our evaluation we motivate heterogeneous sensor network by running the same set of experiments with and without the Stargates. These two setups are referred to as the "heterogeneous" and "single-server" setups, respectively. When the Stargates are *not* used the server communicates to the radio network using the MIB and its NIM. Our objective is to study the performance in terms of the participation of the nodes in the network, system yield of the processed data, and the response (to the `ctrl` controller) of the network. We study the performance by varying the number of Mica2 nodes in the network from one to ten. Over the experiments the layout is kept uniform. The order of switching on the motes follows ascending order of their ids ($m50 - m59$). Thus, in an experiment that uses 8 motes, nodes $m50$ to $m57$ are turned on. The radius of the network is less than 30 feet.

5.2.2 Self-Organized Tree Routing

When the sensor system is turned, on the wireless sensor network self-organizes into tree based on the HTR implementation of the network service FC that runs on each node. The resulting tree formed for a 10 mote network is shown in Figure 6. The tree on the left includes the Stargates in the network and represents a hierarchical heterogeneous network. In the tree on the right, the motes in the network connect to the server using the NIM on the MIB. The layout of the illustrations also indicate the approximate geographical layout of the nodes in the lab. Note the formation of child relationships, which are denser in the single-server setup.

5.2.3 Quality of *Max* Data

The `max` FC provides non-overlapping time window max values. In our program the length of this time window is $30 \times 12 = 360$ms (where 30ms is the sampling interval and 12 is the length of buffers (cf. §4.2) produced by the `accel` device component. Merging values in time windows requires alignment of data in global time windows. Thus, the max data stream is passed through a time-align contract (i.e., a language FC that implements time-align sequencing and buffering). This LFC utilizes a simple flooding based time synchronization protocol implemented through the network SFC of mOS. The quality of the max data depends on the participants in calculating the merged value (observed at the root) that represents the system max (note that the `max` FC also counts the participants used for each windows and sends it in the `max_t` stream). The number of participants can be lower than the nodes in the network due to (1) packet losses, and (2) time misalignment between nodes.

The plot in Figure 7 illustrates the system performance of max-processing in terms of average node participation in each time window over which max was evaluated. The average was calculated based on the count of participants in each max packet recorded over a period of 60 seconds. This represents the quality of max data generated by the macroprogram. We see that with up to five nodes in the network both the heterogeneous setup and the single-server setup provide nearly 100% participation. As the number of nodes increase the single server setup starts to lag. While the heterogeneous system gives nearly a perfect participation till there are 7 nodes in the network. In the worst case it gives almost 90% average participation while a single-server system reduces down to 75% average participation. As the number of nodes in the network increase and geographical span of the network increases more packet losses and time mis-alignment is expected. The use of two Stargates reduces this, because they are geographically close to the motes.

To provide detailed view into the participation distribution for max merging per time window we plot the complete sequence of counts for an $\approx$ 60 seconds run. The results, shown in Figure 8 pertain to a 10 Mica2 node system in the heterogeneous setting.

5.2.4 Response Time

A key feature of our example program is that it uses a controller to conserve resources. High fidelity view of a accelerometer readings is triggered through feedback on detection of "interesting" activity. The time from the point where a triggering message is released from the controller, to the time where the requisite high fidelity data is received at the server is called the data response time. Data response time can be easily and accurately measured at the root node of the tree. Note that the reception of the data includes the time spent buffering (and processing) the data as well. In our example, the data buffer for FFT is set to 120 samples, which translates to

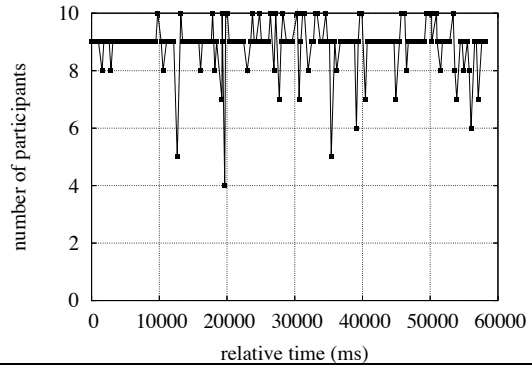

Figure 8. Participation over time per max merge window observed at the root (server) for a network with 10 nodes, 2 Stargates and a server.

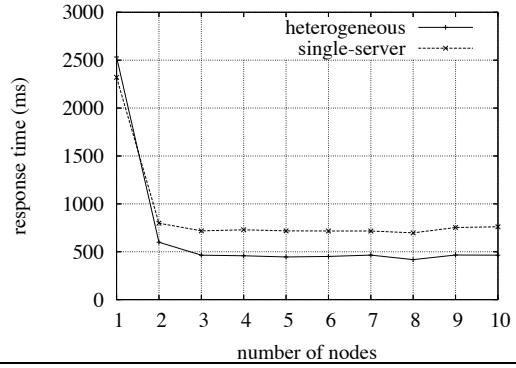

Figure 9. Event response time.

$120 \times 30 = 3600$ms. The data response time less data buffering time is an *estimate* of event response time, which is of interest to us. This estimate of event response includes processing time. It is also affected by packet losses, because the data response is affected due to packet losses. As a metric of system performance we measure the minimum event response time. From a structural engineer's perspective the minimum time is most important because it determines when the first view of the building "activity", in our case frequency response, is observable. The last response (i.e., max response time) response is more or less redundant.

The plot in Figure 9 shows the estimated event response time for the heterogeneous and single-server setups as the number of Mica2 nodes in the network are varied. The results show that irrespective of network size the event response time stays almost constant. For the network without Stargates, the response time is almost 50% higher. This can be attributed to the more reliable and faster data paths offered by the Stargates. One anomaly is the high response times for a single Mica2 node network. After further investigation this was found to be caused by the loss of several initial packets when transmission of packets at a high rate is initiated by a Mica2 mote when it is relatively idle. However, the reason for these initial correlated losses is still under investigation.

The maximum event response time (and even the average) varies highly because some nodes may miss the trigger broadcasted down the tree from the controller. This observation shows the importance of re-enforcing triggers in the network to increase system yield. The controller FC implementation periodically refreshes triggers given continued variations or high values of max data.

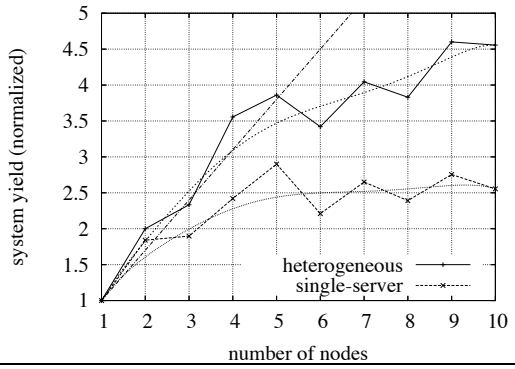

Figure 10. System *frequency response* data yield as the number of nodes in the network increase. Frequency response requires high-fidelity sampling, which results in high contention in the shared radio medium. Trends using linear extrapolation and bezier curves are illustrated.

5.2.5 Yield of High Resolution Data

Our macroprogram application requires generating a frequency response of the building under external stimuli. This requires the performing FFT over high-fidelity accelerometer readings. With ≈ 33Hz sampling a large amount of data is produced, which needs to be transmitted over an un-reliable Mica2 radio network. The use of a large number of nodes in the system enables a redundant and reliable view of the building oscillations. We evaluate the yield of the system as whole for our macroprogram.

When the sensor system is run using a single Mica2 mote, on average, $\approx 91\%$ of expected FFT results are retrieved, as compared to an ideal system. This retrieval rate is used as the base yield of the system (i.e., it represents a yield of 1). The system yield, as the number of nodes are varied, is show in the plot in Figure 10. The results for the heterogeneous and single-server setup are presented. Trends in the plot are illustrated using two bezier curves and a line. The plots show several interesting features. First, the system with Stargates outperforms the system without Stargates even when the number of nodes is low. This is because the Stargates are closer to motes, and hence, prove to be more reliable specially when data rates are high. Second, the system with Stargates initially gives an almost linear scaling (as illustrated by the line) as the number of nodes increase. This is in agreement with the linear increase in the number of nodes in the network. Third, after the number of nodes in the network grows beyond 5 the performance of both setups deteriorate badly. We attribute this to increased interference in the shared radio channel. We draw this conclusion from the observation that if $m55$ to $m58$ and $sg1$ are moved away from the rest of the nodes the performance of the heterogeneous setup increases (alternatively, the separation can be achieved by reducing radio power). This provides a practical lesson on the performance implication of radio interference and the need to control radio range. Considering, the yield of 3.5 with a network of size 5 (i.e., $91\% \times 3.8 \div 5 \approx 70\%$ reliability) believe that the performance of the system makes it practically feasible for our application domain.

The ease of development of this complex application and its robust performance demonstrates the prowess of COSMOS as an architecture for macroprogramming sensor network systems. Our evaluation also demonstrates the performance gains due to use of heterogeneous devices—a feature that the mPL macroprogramming model and the underlying mOS synergetically enable.

5.3 Reprogramming

COSMOS allows reprogramming of applications over-the-air. To launch an application, each node needs a subgraph (based on the node's capability) of the IA. The mPL compiler generates *m-messages* for each FC and device component instance in the application IA graph (including the LFCs and SFCs that are transparently spliced into user provided IA description). m-messages describe the output connections of an FC or device component. Each m-message also contains the machine capability constraint associated with the component that it describes. Thus, this message is only used by machines that have the requisite capability. For this reason, multiple messages may also be produced for a single FC (e.g., for the program in Figure 1, `max` FC connects to the network SFC on motes, while it connects to the `disp2` driver at the server). Each m-message is encapsulated in a network message, which is transmitted during reprogramming. Thus, the fundamental overhead, due to COSMOS architecture, for application reprogramming is the size of m-messages. Table 1, provides the size of m-message for the application in Figure 1. Note that the sizes are very small and usually fit in a single network packet (even over the radio). The size of an m-message increases as the number of output connections of a component increase.

Component name	Machine capability	Size bytes
accel	`MCAP_ACCEL_SENSOR`	13
cpress	`MCAP_ACCEL_SENSOR`	13
thresh	`MCAP_ACCEL_SENSOR`	11
max	`MCAP_ACCEL_SENSOR`	11
max	`MCAP_UNIQUE_SERVER`	13
max	`NOT(MCAP_ACCEL_SENSOR) & NOT(MCAP_UNIQUE_SERVER)`	11
fft	`MCAP_FAST_CPU`	11
ctrl	`MCAP_UNIQUE_SERVER`	11
disp1	`MCAP_UNIQUE_SERVER`	11
disp2	`MCAP_UNIQUE_SERVER`	11
net(SFC)	`MCAP_ACCEL_SENSOR`	16
net(SFC)	`MCAP_UNIQUE_SERVER`	21
net(SFC)	`NOT(MCAP_ACCEL_SENSOR) & NOT(MCAP_UNIQUE_SERVER)`	17
buff-len(LFC)-fft	`MCAP_FAST_CPU`	11
buff-time(LFC)-max	`NOT(MCAP_ACCEL_SENSOR)`	11

Table 1. Size of m-messages for components of the application in Figure 1.

In addition to m-messages an application initialization message and an application start message is also transmitted. The application initialization message is 7 bytes long and summarizes the IA sub-graph, for each machine capability type that the compiler discovers. This message contains the number of device components and FCs that the node must expect for its IA sub-graph. Application initialization message precedes the m-messages. The application start message, which is 4 bytes long, signals the start of an application. Multiple independent applications can be run concurrently. As the IA-sub graph is kept in data RAM no power overhead due to writing to external flash, or instruction ROM is spent. To stop an application, identified by an id, the user gives a command to the server node, which transmits a 4 byte application stop message. On receiving the message a node, stops running components and drivers, deletes its IA sub-graph. Note that in the current implementation an application must be reprogrammed as a whole (i.e., to modify an application, the user must stop and reinstall it). We plan to change this in the future to include difference based updates.

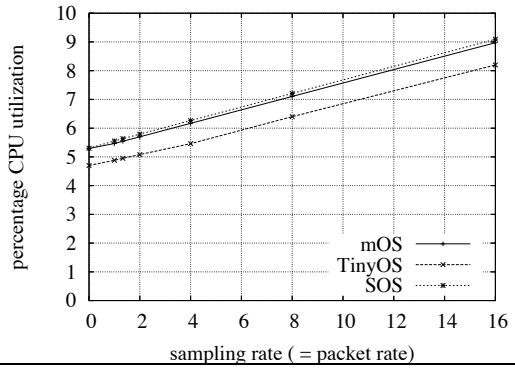

Figure 11. CPU utilization of COSMOS, TinyOS and SOS for a simple data sampling and transmission application.

For transmission of m-messages, machine capability based flooding (i.e., a packet is broadcasted only if the descendents of a node include the requisite capability) is used. Once a node receives the start message it knows how many messages to expect and uses timer based NACKS to request messages from neighbors. To reprogram a 13 node network with (10 Mica2 nodes, 2 Stargates, and 1 server) on average 80 messages over the mica2 radio network, and 25 messages over the 802.11 network were communicated. This is insignificant compared to the amount of data traffic even in 10s of hours of low duty cycle operation. Note that the network protocol overhead is orthogonal to the design of COSMOS and details are skipped here. Other protocols, e.g., Trickle [11], can be easily implemented as a service FC. Updating or launching applications may require updating the binary of FCs. This mechanism has been borrowed from SOS operating system and a discussion on its overhead can be found in [8].

5.4 Micro Benchmarks

In this section, we present micro-benchmarks to evaluate the footprint of mOS on resource constrained nodes. We use Mica2 as the hardware platform. We evaluate the processing costs of mOS using Avrora [23]. Avrora provides a faithful cycle accurate simulation of the Mica2 platform. Monitors provided by Avrora can be used to accurately profile executions, track hardware I/O (e.g., packet transmission) and monitor CPU utilization. The simulated time of all benchmarks is 200 seconds.

5.4.1 CPU Utilization

We study the CPU utilization using a simple remote sensing macroprogram that takes a sensor reading, which is packetized and transmitted over the radio. This application stresses the OS as no application processing is involved. A similar application implemented for TinyOS and SOS allows comparison with these platforms. TinyOS is a low-overhead operating system popularly used for sensor networks. SOS allows use of extended features (beyond TinyOS) such as separation of the OS and the application, and dynamic memory, similar to mOS.

Figure 11, shows the CPU utilization of COSMOS and SOS operating systems with varying sampling rates. In this experiment, we do not use any sample buffering, hence, the packet rate is the same as the sampling rate. The results show that when no sampling is performed, SOS and mOS have similar overheads. With increasing sampling rates, CPU utilization on both systems increases linearly, while CPU utilization on mOS is slightly lower. TinyOS performs better than mOS by an absolute difference of 1%, which is low. However, nither TinyOS nor SOS allow macroprogramming. These

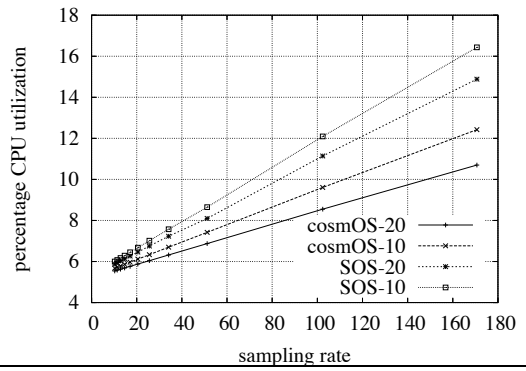

Figure 12. Effect of buffering data samples on CPU utilization. Buffer sizes of 10 and 20 samples are used.

#	Start	End	Cycles
1	Timer fire	Driver data req	960
2	Driver data req	Data rcvd	993
3	Data rcvd	Sched FC	526
4	Sched FC	Start FC exec	397
5	Start FC exec	Inside FC	198
6	Send net call	-	622

Table 2. Time elapsed between various operations of COSMOS on Mica2.

result shows that COSMOS provides a more powerful macroprogramming model without sacrificing performance.

Application developers for COSMOS can request device ports to buffer samples. The performance benefit of this strategy is illustrated in Figure 12. Two buffer sizes, 10 and 20, were used (thus, packet rates are reduced by a factor of 10 and 20, respectively). The application for SOS was modified to buffer at the "sensor driver module" (which provides the only opportunity for application developers to affect sample buffering). We note that both platforms benefit from buffering, however, COSMOS offers much more significant savings with increasing sample buffering. The limit of 171sps is imposed because the timer subsystems on both systems imposes a lower limit of 5ms firing interval.

5.4.2 Detailed Processing Costs

Using the profiling features of Avrora, we study the low-level footprint of mOS for our remote sensing application (without buffering). Table 2 summarizes the average elapsed cycles for different operations in the execution of the application. The sampling rate is set to 8sps (thus, the system is under suitably low load), which yields 1600 points in each data set. The results show an average over these points for each set.

Item 1 is the number of cycles elapsed from the instant the timer fires to the scheduling of the device port, which is observed to be 960 cycles. This is equal to 0.13ms, given the processor speed of 7.3728MHz. This operation involves timer delta queue operations, enqueue of the component execution request to the scheduler queue, and finally the dispatch leading to the execution of the device component. On execution, the device component allocates memory for the requested data object and calls the hardware data request function.

Item 2 corresponds to the cycles elapsed from this request till the time data is made available to be enqueued to the next FC in the IA. This time includes a request to the ADC, firing of the ADC interrupt, and depositing the data in the data object memory. Once the object is ready it leads to the scheduling of the next FC. Item 3 shows this cost. Item 4 shows the elapsed cycles from the time of enqueuing of the request to execute an FC on the scheduler to the scheduler dispatch leading to the execution of the FC. In this experiment, the scheduler queue is empty, hence, this approximates the actual cost of scheduling and dispatching a request using the system scheduler. If data buffering is requested, the cost of tasks in items 3 and 4 is reduced to once per buffer, instead of once per each sample. Further, memory allocation (of the data object) in item 1 is also performed once per buffer. Item 5 shows that the cost of preparing parameters to be passed to the handler of the FC and its invocation is only 198 cycles (or 26.8μs). If the FC has multiple inputs for which data is ready, 153 cycles are used for preparing each additional invocation.

Finally, we show the cost of a system call to transmit a data object in item 6. Recall, that this call is made only from network service FC. This system call involves allocating a packet header, linking the object (to be transmitted) to this header and a call to the radio stack driver. In all, the time elapsed from the timer event to the radio is approximately $3,800$ cycles or $\approx$ 0.5ms. This value is small, relative to the data processing operations. We conclude from these micro-benchmarks that mOS provides a low footprint environment for macroprogram execution.

5.5 Cost of FC Chains

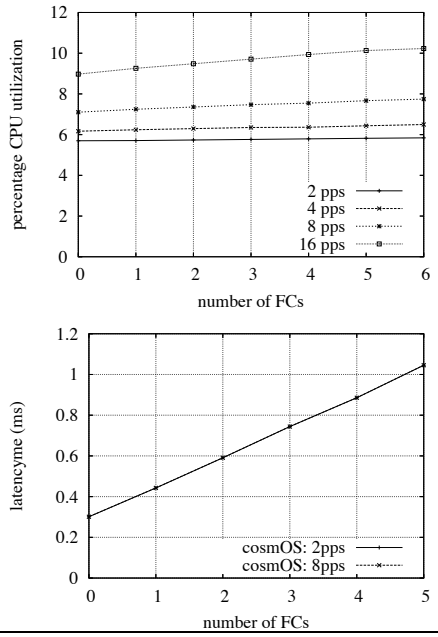

Figure 13. Effect of increasing the number of FCs on the data path.

The design of COSMOS promotes the use of small specialized FCs rather than large FCs (both in terms of size and processing time). Figure 13 illustrates the effect, in terms of CPU utilization and data latency, of increasing the number of FCs on the data path of the remote sensing application. The application used in earlier sections had 1 FC (in addition to the network service FC). Note that 0 FCs means that the sensor driver port is directly connected to the network service FC. The additional delay added to the data path is less than 0.2ms per FC, and does not vary with increasing packet rates. At low packet rates, the added processing due to FCs in the data path increases the CPU utilization negligibly. At higher rates the increase in CPU utilization is small, e.g., at 16pps, adding 5 FCs increases the overall CPU utilization by an absolute value

of 1% in comparison to the CPU utilization when 0 FCs are used, given the same packet rate. We also evaluate the effect of number of FCs on the path of data being forwarded through the node. The results (not shown) reflect identical qualitative behavior.

6. Related Work

TinyOS [9], together with NesC [6], is among the most popular platforms for sensor network development. It represents a holistic approach, where the OS is a part of the application. Therefore, developing applications in TinyOS requires handling low-level issues. To ease application development in TinyOS, high-level languages such as SNACK [7], Mottle [20], and TinyScript [20] can be used. Unlike macroprogramming, though, these languages rely on explicitly programming inter-node messaging to encode system behavior.

In contrast to TinyOS, SOS [8] separates the OS and the application. Applications developed as message handling modules can be loaded at runtime providing ease of development and reprogramming. However, application development involves explicit messaging, which is different from COSMOS. This may result in hidden intermodule dependencies and related failure modes. COSMOS, on the other hand, uses functional components (FCs), which are standalone modules that can be reused across applications. Through composition of FCs, COSMOS allows direct specification of aggregate system behavior. Furthermore, the macroprogramming model of COSMOS provides vertical integration over a heterogeneous sensor network unlike TinyOS and SOS.

Similar to COSMOS, MagnetOS [12] views the entire sensor network as a single entity and provides a unified Java VM that transparently encompasses the network nodes. However, MagnetOS is targeted at a different application domain with more powerful machines (such as laptops and PocketPCs).

Regiment [19] is a high-level functional macroprogramming language, based on the idea of manipulating region streams that are spatially distributed, time-varying collections of node states. Regiment programs are compiled down to an intermediate language called Token Machine Language (TML) [18]. Execution of TML-based programs involve message passing to coordinate computation across nodes. Hood [27] and abstract regions [25] provide geographic neighborhood based communication and programming abstractions that use spatially distributed shared variables. While well suited to specific application domains, the applicability of these systems as general sensor network development models is still the subject of current investigations. These languages involve sharing state and coordinating computation in a distributed environment. In contrast, COSMOS is data-centric and provides dataflow through a graph of data processing functional components.

Semantic Streams [26], TAG [15], TinyDB [14], and Cougar [28] view the sensor network as a stream database and determine global system behavior using queries. However, these are not macroprogramming environments as such, rather, they represent a macroprogram application that allows generic (e.g., SQL based) queries. We believe that various generic or high performance domain-specific query based stream processing systems can be rapidly implemented using macroprogramming in COSMOS.

The idea of using graphs of processing components to develop complex and robust applications is well known in systems literature. UNIX System V STREAMS [22], FreeBSD Netgraph subsystem, Click router [10], Scout OS [17], and Seda [24] are examples of high performance systems that exploit this paradigm. COSMOS extends the idea of connected graphs of functional components to provide a low-footprint platform for macroprogramming sensor networks.

7. Concluding Remarks

COSMOS, comprising of mPL and mOS, offers a novel platform for macroprogramming heterogeneous sensor networks. The associated macroprogramming model allows explicit specification of distributed system behavior using interaction graphs, which can be statically checked for correctness. Additionally, mPL provides contract expression based rich abstractions to develop robust, scalable and adaptive macroprograms for self-organized sensor network environments. COSMOS is available for public release.

References

[1] Bowen labs. `https://engineering.purdue.edu/CE/BOWEN/Facilities`.

[2] Crossbow Inc. `http://www.xbow.com/wireless_home.aspx`.

[3] AWAN, A., JAGANNATHAN, S., AND GRAMA, A. Verifiable high-level abstractions for macroprogramming sensor networks. Unpublished manuscript.

[4] BORGERDING, M. Kiss FFT. `http://sourceforge.net/projects/kissfft/`.

[5] FINDLER, R., AND FELLEISEN, M. Contracts for higher-order functions. In *Proc. of ICFP '02* (October 2002).

[6] GAY, D., LEVIS, P., VON BEHREN, R., WELSH, M., BREWER, E., AND CULLER, D. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. of PLDI '03* (June 2003).

[7] GREENSTEIN, B., KOHLER, E., AND ESTRIN, D. A sensor network application construction kit (SNACK). In *Proc. of SenSys '04* (November 2004).

[8] HAN, C.-C., RENGASWAMY, R. K., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. SOS: a dynamic operating system for sensor networks. In *Proc. of MobiSys '05* (June 2005).

[9] HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. System architecture directions for networked sensors. In *Proc. of ASPLOS-IX* (November 2000).

[10] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[11] LEVIS, P., PATEL, N., CULLER, D., AND SHENKER, S. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of NSDI '04* (March 2004).

[12] LIU, H., ROEDER, T., WALSH, K., BARR, R., AND SIRER, E. G. Design and implementation of a single system image operating system for ad hoc networks. In *Proc. of MobiSys '05* (June 2005).

[13] LIU, T., AND MARTONOSI, M. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proc. of PPoPP '03* (June 2003).

[14] MADDEN, S., FRANKLIN, M., HELLERSTEIN, J., AND HONG, W. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems 30*, 1 (March 2005), 122–173.

[15] MADDEN, S. R., FRANKLIN, M. J., HELLERSTEIN, J. M., AND HONG, W. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. of OSDI '02* (December 2002).

[16] MEYER, B. Applying design by contract. *IEEE Computer 25*, 10 (October 1992), 40–51.

[17] MOSBERGER, D., AND PETERSON, L. L. Making paths explicit in the Scout operating system. In *Proc. of OSDI '96* (October 1996).

[18] NEWTON, R., ARVIND, AND WELSH, M. Building up to macroprogramming: An intermediate language for sensor networks. In *Proc. of IPSN '05* (April 2005).

[19] NEWTON, R., AND WELSH, M. Region Streams: functional macroprogramming for sensor networks. In *Proc. of DMSN '04* (August 2004).

[20] P. LEVIS ET. AL. Maté: Programming Sensor Networks with Application Specific Virtual Machines. `http://www.cs.berkeley.edu/~pal/mate-web/`.

[21] RATNASAMY, S., KARP, B., SHENKER, S., ESTRIN, D., GOVINDAN, R., YIN, L., AND YU, F. Data-centric storage in sensornets with ght, a geographic hash table. *Mobile Networks and Applications 8*, 4 (2003).

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

[23] TITZER, B., LEE, D., AND PALSBERG, J. Avrora: Scalable sensor network simulation with precise timing. In *Proc. of IPSN '05* (April 2005).

[24] WELSH, M., CULLER, D., AND BREWER, E. SEDA: an architecture for well-conditioned, scalable internet services. In *Proc. of SOSP-18* (October 2001).

[25] WELSH, M., AND MAINLAND, G. Programming sensor networks using abstract regions. In *Proc. of NSDI '04* (March 2004).

[26] WHITEHOUSE, K., LIU, J., AND ZHAO, F. Semantic Streams: a framework for composable inference over sensor data. In *Proc. of EWSN '06* (February 2006).

[27] WHITEHOUSE, K., SHARP, C., BREWER, E., AND CULLER, D. Hood: a neighborhood abstraction for sensor networks. In *Proc. of MobiSys '04* (June 2004).

[28] YAO, Y., AND GEHRKE, J. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record 31*, 3 (September 2002), 9–18.

[29] YARVIS, M., KUSHALNAGAR, N., SINGH, H., RANGARAJAN, A., LIU, Y., AND SINGH, S. Exploiting heterogeneity in sensor networks. In *Proc. of INFOCOM '05* (March 2005).