Building up to Macroprogramming: An Intermediate Language for Sensor Networks

Ryan Newton and Arvind
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139
Emails: {newton, arvind}@mit.edu

Matt Welsh
Electrical Engineering and Computer Science
Harvard University
Cambridge, MA 02138
Email: mdw@eecs.harvard.edu

***Abstract*— There is widespread agreement that a higher level programming model for sensor networks is needed. A variety of models have been developed, but the community is far from consensus. We propose an intermediate language to speed up the exploration of this design space. Our language, called the Token Machine Language (TML) can be targeted by compilers for higher level systems. TML provides a layer of abstraction for a lower-level runtime environment, such as TinyOS.**

TML is intended to capture *coordinated activity* in a sensor network. Notable features of TML are its atomic action model of concurrency, and its unification of communication, control, and storage around the concept of a token. Tokens are small objects, typically under a hundred bytes, and can be disseminated across the network. A token causes computation upon its arrival at a site by invoking a *token handler*. The effect of the computation is to atomically change the token's own state as well as the state of shared variables at the site.

I. Introduction

A breakthrough in programming technology for ad-hoc sensor networks may help them achieve widespread acceptance across commercial and scientific spheres. A number of ongoing projects are developing better programming tools for sensor networks. These include virtual machines [15], high-level programming abstractions [3], [20], and query engines [16], [23]. One approach, which we call *macroprogramming*, would allow application designers to write code in a high-level language that captures the operation of the sensor network as a whole. The global program could then be compiled into a form that executes on individual nodes.

In order to develop high-level languages that compile into node-level programs, it would be extremely valuable to define a common intermediate language. Without one, we are forced to build monolithic implementations with a language such as NesC. We need an intermediate representation to abstract away the details of concurrency and communication while capturing enough detail to permit extensive optimizations by the compiler.

In this paper, we propose such an intermediate language for sensor networks, called the Token Machine Language (TML). TML is based on a simple abstract machine model, which we call Distributed Token Machines (DTMs). Distributed Token Machines provide an execution and communication model based on *tokens*. Communication happens through token messages, which are typed messages containing a small payload. Tokens are associated with *token handler*s that are executed upon reception of a token message (either locally or from a radio message). Tokens are akin to Active Messages [19], although DTMs provide additional facilities for structuring execution, concurrency, state management, and communication that make this model attractive as a compilation target for sensor network applications.

Our goal is to define an intermediate language for sensor network programming that:

1) Provides simple and versatile abstractions for communication, data dissemination, and remote execution.
2) Constitutes a framework for network coordination that can be used to implement sophisticated algorithms such as leader election, group maintenance, and in-network aggregation.

While not an explicit goal, it turns out that TML is also usable by humans as well as code generators , as should be evident in the example programs of Section V.

TML is meant to be lightweight in every respect. It must map efficiently onto the event-driven semantics of existing sensor network operating systems, such as TinyOS. TML also must be versatile enough to construct a wide range of higher-level systems, while simultaneously masking the complexities of the underlying OS and runtime environment. These goals differentiate TML traditional intermediate languages, such as the Java Virtual Machine and CLI, which are have been designed for portability and safety. Instead, we draw on the lineage of systems such as the Threaded Abstract Machine (TAM) [6], which aims to provide an appropriate level of granularity to achieve abstraction without sacrificing performance or versatility.

TML provides a unified abstraction for communication, execution, and network state management based on tokens. Specifically, all communication in TML is accomplished by dissemination of tokens. Upon receiving a token, a node atomically executes the associated token handler, which uses and modifies the stored form of the token, schedules new tokens, and kills existing tokens. This approach allows an application to refer to the tokens as well as to the set of nodes holding a particular token as semantically meaningful units. For example, tokens make it straightforward to implement network abstractions, such as *gradients*, which involve flooding a token throughout all or part of the network and constructing a spanning tree pointing back to the origin of the gradient. Gradients are a common routing model in sensor networks [14], [16] which can be implemented in TML in a manner that meshes with its token abstraction.

In Section III of this paper we present the semantics of the Distributed Token Machine model. The DTM is an abstract machine, not a complete executable language. Thus in Section IV we will present the concrete language TML — our realization of the DTM model. In Section V, we demonstrate how to use TML in writing several simple applications, including a distributed event detector and a decentralized leader election algorithm. Finally, we will discuss our prototype implementation of TML on top of TinyOS, showing that the TML abstraction introduces little overhead in terms of code sizes, and only modest overhead in terms of RAM usage and execution speed.

II. Related Work

There have been a variety of attempts to raise the bar for sensor network programming, including middleware services, communication abstractions, and programming models.

Of these, the Maté system is most similar to TML. It provides a platform for higher-level languages, but does so through a virtual machine rather than a compiled intermediate language. The project focuses on application specific VM extensions, safety guarantees, and energy-efficient dynamic reprogramming. However, the overhead of bytecode interpretation makes intensive computation prohibitive. TML's approach is to use more efficient compiled code, but at a cost of larger binaries. Further, safety can be guaranteed by the macroprogram compiler, not the run-time system, or even the intermediate language — TML itself is not type-safe, for example.

Maté hosts high-level programming systems through application specific VM extensions. For example, TinySQL is a query processing system built on a specialized Maté VM. Another specialized VM incorporates the abstract regions communication model of Welsh and Mainland [20]. Maté differs from TML in that it provides only low-level radio communication directly within Maté, and uses application-specific opcodes — essentially a foreign function interface into other TinyOS components — to expose new communication primitives such as abstract regions. In contrast, TML provides a coordinated communication and execution architecture, based on tokens, that is used to build up new communication abstractions.

The Impala middleware system enables application modularity and network reprogramming, while MagnetOS [2] aims to provide automated assignment of Java components to parts of an ad-hoc network (but has not been scaled down to resource-constrained sensor networks). Middleware systems such as these share goals with TML, but they focus on providing a rich array of run-time services. TML, on the other hand, requires very little in the way of run-time support; its core implementation is a few hundred lines of NesC code. Instead, TML is focused on defining the appropriate semantics useful for compilation.

A number of recent projects have proposed novel communication abstractions for sensor networks. For example, Spatial Programming [4] uses Smart Messages to provide content-based spatial references to embedded resources. In this system, the programmer may refer to the first available camera in a given (predefined) spatial region. Alternatively, Blum *et al.*, 2003 describes an "entity maintenence" abstraction that exposes tracked targets as first-class language objects which serve as communication endpoints. Other related communication abstractions include abstract regions [20], Hoods [21], DIFS [10], and DIMENSIONS [8].

A few communication models in particular had an influence on TML. The general purpose gradient and aggregation interface of TML (Section V-A) are similar to those in Directed Diffusion [14] and SPIN [12], which are paradigms for source-sink communications over named data. Also, work on Amorphous Computing [1] explores in depth the applications of gradients. TML is also heavily inspired by Active Messages (AM), which were originally conceived as a mechanism for efficient message passing in parallel architectures [19]. The original AM work focused on integrating communication and execution for constructing parallel applications. AM has found a new home in TinyOS [13], although it is used there primarily as a radio message format, rather than for introducing specific semantics for the execution of message handlers. For example, the TinyOS variant of AM does not specify how messages interact with the link and network layers of a protocol stack.

Additionally, several high-level languages have been proposed for programming sensor network applications, including Mottle [22], TinyScript [22], SNACK [9], and the variants of SQL found in Cougar [23] and TinyDB [16]. The goal of TML is to define an appropriate intermediate language to which these systems can compile down. Without a common abstraction, sharing functionality is more difficult and complexity less seperable. For example, the TinyDB system is monolithic and includes a wide range of functionality: spanning tree formation, query dissemination, query optimization, and in-network aggregation.

Our prior work on Regiment [17] defines an alternative macro-programming language based on functional reactive programming. Regiment represents sensor nodes as streams of data that can be grouped into regions for the purpose of in-network aggregation or detecting events. Because of its high level nature, the semantic gap between Regiment and a node-level language like NesC is large, making the task of compilation daunting. It was in the course of implementing a Regiment compiler that we created the TML intermediate language. We return to a discussion of Regiment and its compilation to TML in Section V.

III. Distributed Token Machines

The Distributed Token Machine (DTM) is an abstract model for computation in a dynamic, asynchronous, ad-hoc network with stopping failures and message losses. The DTM model structures concurrency, communication and storage according to the architecture pictured in Figure **??**. However, it makes few assumptions about which scheduling algorithm is used or about the base language used to describe the actions performed by each token handler. The Token Machine Language introduced in the next section is one realization of the DTM model using a concrete handler language similar to a basic subset of C.

A. Execution Model

In the DTM execution model, each node in the network holds some number of *tokens* as well as a single fixed-size shared memory. Each token has an associated *token handler* that is executed upon receipt of the token in a message. The token handler takes arguments that it receives from the payload of the message carrying the token. Each token also has a private memory, which is a fixed-size piece of state that may only be accessed by the associated token handler. The stored token along with its private memory is referred to as a *token object*. In OOP terms, each token object can be thought of as an object with only private fields and only one method. Each handler executes atomically and in a bounded amount of time, which simplifies memory consistency issues and makes precise scheduling possible. This model is restrictive, but its simplicity is a boon to compilers and analysis tools working with the architecture. The DTM model's selling point is its combination of expressiveness and simplicity.

DTMs have only the core ingredients necessary to build up higher level programming features. For, example, it would be straightforward to map multiple token "methods" down onto the one supported handler action. Similarly, issues of memory scope and protection can be compiler controlled. Most importantly of all, a compiler can break a long sequence of coordinated actions into a set of token handlers that interact in predictable ways, both inside nodes and across the network.

Figure 1 depicts the structure of a sensor node executing the DTM model. A node in the network consists of a heap storing local token objects (the token store), a scheduler that processes incoming token

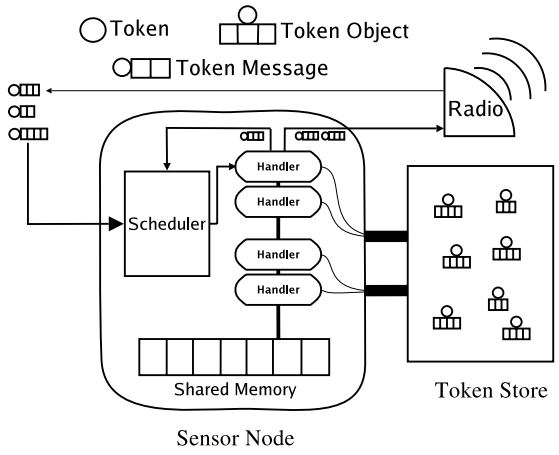

Fig. 1. The structure of a node in the DTM model. Token messages are transmitted on network channels. When received, they morph into their persistent form: token objects.

messages, and a collection of handler actions that are computable by the node. Again, just as the DTM model does not fix the base language used for handlers, the scheduling algorithm also remains implementation-specific.

Token messages are the form tokens take when traveling over network channels. The token name is paired with an associated payload. As in Active Messages, the token-name is at the start of each message. (Also, in the DTM model we allow token names to be encoded as data and transfered in messages or stored in memories.) Once a message comes through the scheduler, the token name directs it to the corresponding handler. If the corresponding token object is not already present in the token store, its memory is allocated and initialized (to zero) before the handler begins running. The handler consumes the message payload and executes atomically, possibly reading and modifying the token object's private memory and the node's shared memory in the process.

B. Handlers and their Communication Interface

Executing token handlers must post new messages. Thus, they must interact with the scheduler. We still do not wish to specify exactly what language is used for describing handlers — we do not care, for example, what concrete syntax or data types are used — but we do wish to specify the interface through which the handler must interact with the rest of the system. Token handlers must use the below operations to access or modify the token store and the scheduling queue.

- **schedule**($\mathcal{T}_i, priority, data \ldots$)
- **timed_schedule**($\mathcal{T}_i, time, data \ldots$)
  The **schedule** operation inserts a token message in a nodes' local scheduling queue. $data \ldots$ is the payload of the token message. **schedule** is non-blocking and returns immediately. **timed_schedule** is a version of **schedule** that causes a message to execute on the local node at a precise time — after a given number of milliseconds. This time requirement overrides whatever other strategy the scheduler uses to order messages. But in general the scheduler may use an implementation specific algorithm for scheduling incoming messages. The acceptable values for the $priority$ argument are also implementation specific and are only relevant insofar as they affect that particular scheduler. The scheduler even has permission to drop remotely received messages — legal because the DTM model is defined in terms of lossy channels. However, the scheduler is expected to follow certain rules. It must respect the relative order of all token messages produced by during the execution of a single atomic token handler. Thus two consecutive **schedule** commands in a handler will always have their order preserved.
- **bcast**($\mathcal{T}_i, data \ldots$) **bcast** is a version of **schedule** which, instead of handing the token message off to the local scheduler, broadcasts it to the radio neighbors of the node running the executing handler. The message may subsequently be received by one neighbor, several neighbors, or none, and the DTM model does not assume ACKs in the communication protocol. This message loss must be dealt with by the executing token machine. At its core, DTMs provide only simple single-hop broadcast, nothing more. More complex communication primitives are built up from there (see Section V-A).
- **is_scheduled**($\mathcal{T}_i$)
- **deschedule**($\mathcal{T}_i$)
  is_scheduled and **deschedule** allow query and removal of token messages that are waiting in the scheduler. **is_scheduled** only reports the presence or absence of a given token — not the number or timing of token messages affiliated with that token. Similarly, **deschedule** removes all messages with a given token name.
- **present**($\mathcal{T}_i$)
- **evict**($\mathcal{T}_i$)
  DTM does not specify how handlers interact with their own private memories — it is expected that they will have appropriate load and store operations — but it does specify the interface into the nodes' token store as a whole. **present** queries the local node's token store for the presence of a token of a given name. **evict** removes the token, if present. When a token name T_i is evicted, the corresponding token object frees its private memory — no record of the token's presence is kept. If a token handler evicts itself while executing, the eviction occurs upon the handler's completion.

IV. Token Machine Language

In this section we describe a realization of the DTM model — the Token Machine Language (TML). The DTM model provides the execution model; TML fills in set of basic operators and a concrete syntax for describing handlers. The language used in TML for the bodies of handlers is a subset of C extended with the DTM interface described above (schedule, evict, etc). This subset disallows data pointers, allows only fixed-length loops, and uses the scheduling of tokens as its only procedure call mechanism. Conditionals are permitted, but they can make the execution time of a handler undecidable (but still bounded).

Named fields such as `(int16 x, int32 y)` replace the undifferentiated blocks of bits described in the DTM model — both for the handler arguments (the token message payload) as well as for the token object memories and the shared memory. Data in the token object's private memory can be statically allocated by declarations of the form: `stored int x, y;` (initialized to zero). Data shared by all tokens is declared as `shared int z;`. TML also allows trailing arguments to handlers to be omitted with the understanding that they will take on a zero value. This makes some scheduling and broadcasts of token message more efficient. Here is a sample of nonsense code showing what a token declaration looks like.

```
shared int s;

token Red (int a, int b) {
  stored int x;
  if ( present(Green) )
    x = 39;
  else {
    x += a + b;
    timed_schedule Red(500, s, a);
  }
}
```

The current prototype TML implementation targets the NesC/TinyOS environment. The DTM interface (`bcast`, `schedule`, etc) becomes a TinyOS module (DTM.bcast, DTM.schedule, and so on). Handlers are compiled into individual NesC commands. Our TML compiler provides certain guarantees for generated NesC code; it will conform to the DTM execution semantics — handlers will terminate, programs will not crash motes, and invalid memory references are impossible due to the lack of pointers in TML. There is, however, nothing to stop the user from bypassing our TML compiler and manually writing code against our DTM library. They would then be responsible for respecting the constraints of the DTM model, or at least breaking them in controlled ways.

Token Namespace in TML: The token store is the *only* place for dynamic memory allocation in the system, which does not even have a call-stack. In order for programs to make best use of the token store, they need a way to create an arbitrary number of token names rather than just those that occur in the program text. Thus we allow unique "subtokens". If a program consists of token declarations for tokens {`Red`, `Green`, `Blue`}, the user may also reference `Red[11]` and `Green[32]`. All subtokens of Red use the same handler, but have their own token object in the store and thus keep their own private memories.

Handlers then need a way to refer to the number of the subtoken currently executing. For this reason we allow a syntax similar to component parameterization in NesC. When declaring a token handler we write "`token Blue[id]() { ...`". The variable `id` refers to the numeric index of the subtoken currently invoked. Calling (scheduling) a token without a subtoken index is the same as using index 0.

Subtokens are analoguous to constructing multiple *instances* of a token "class". Subtoken indices are thus a form of pointer in TML, but one that uses a consistent virtual address space across all nodes in the network. They can be used to allocate variable amounts of storage, or as we will see in Section V-A, to keep gradients from overlapping.

A. Returning Subroutines

As our first example of systematically building abstractions on top of TML, we will now add subroutine calls with return values. These are a staple of normal procedural programming. But the DTM equivalent would be **schedule** operations that carry a return value. We must exclude these from the DTM model to keep our atomic actions small and fast. (Besides, our model has only one dynamically allocated structure, the token store; a call-stack would constitute another.)

As a user, however, one would like a token handler to be able to invoke another handler get back a return value. We enable this feature by building returning handler-calls on top of core TML using a continuation passing style (CPS) transformation. This is our answer to the issue of split-phase vs. blocking operations. We circumvent the issue by having *implicitly* split-phase calls. The user simply uses *subcall* as below, and the CPS transformation splits their handler at that point, producing a pre-subcall and post-subcall handler. The programmer must understand that when they use this facility they may break the atomicity of their handler — they are really using syntactic sugar for multiple handlers. This could require freezing all the live variables at the subcall-point, storing them in the continuation token object, and restoring the context again in the post-subcall handler (the continuation token handler), which poses efficiency concerns. As an example, consider the following simple code snippet.

```
token int Red(int a) {
  stored int y = 0;
  schedule Blue(4);
  y = subcall Green(3);
  bcast Red(a);
  return y;
}
```

The token handler for `Red` is transformed so that it stops at the subcall to `Green`; see below for the resulting TML code. At the stopping point, `Red` allocates a continuation object in the token store. The continuation is given a unique name (e.g. `RedK`) and associated with a new compiler-generated token declaration containing the code truncated from `Red`'s handler. This continuation must be invoked somewhere; thus the CPS transformation requires that every handler called via "subcall" take an extra argument naming a continuation to invoke on its return value. Thus, when `Green` is called it is passed the (sub)token name of its continuation (`RedK[...]`), which is equivalent to a pointer to that continuation.

The generated code below is unpleasant because it must invoke the continuation handler in two different modes: first, to allocate a continuation object in the token store; and second, to invoke the continuation. The INVOKE call to `RedK` takes advantage of the fact that the token message payload is really just a fixed size buffer, and omits unused trailing arguments (which become zero). The unsightliness of this code is acceptable because it is automatically generated and not intended for human consumption.

```
token int Red[id](int a) {
  stored int y = 0;
  schedule Blue(4);
  schedule RedK(ALLOC, a, y);
  schedule Green(3, RedK[id]);
}

token RedK[id](int mode, int[2] freevars) {
  stored int a, y;
  if (mode == ALLOC) {
    a = freevars[0]; // captured a
    y = freevars[1]; // captured y
  } else if (mode == INVOKE) {
    y = freevars[0]; // returnval
    bcast Yellow(a + y);
}

token Green(int x, tokname k) {
  ...
  returnval = ...;
  schedule k(INVOKE, returnval);
}
```

Performing a CPS transformation on TML is straightforward because of TML's simplicity and clear semantics. Doing the same for general NesC code would be quite difficult. First, NesC has multiple execution contexts: tasks and events. Second, TinyAlloc is not an convenient mechanism for dynamically allocating and invoking continuation objects.

CPS is well studied in the literature [7], [11], [18]. There are a number of optimizations that can make it more efficient, especially in the case of TML where we can perform whole program analysis. Some techniques minimize the number of continuations created and the circumstances in which they must allocate memory. Further, in the case of TML we can optimize the subcall abstraction layer without breaking the core DTM semantics in the following way. We permit the implementation to choose "direct calls" (i.e. NesC's "call") to *non-recursive* subroutines, rather than going through the scheduler. This is similar to procedure inlining. In our case, since the execution time of each token handler is bounded by a statically known quantity, we can compute an upper bound on the time cost of a non-recursive subroutine call. We then choose to "inline" or not based on the constraints of the scheduling algorithm. For example, we may simply set a maximum desirable atomic action duration, and "inline" subcalls up to that maximum action duration.

Invoking split-phase TinyOS operations: Many operations in TinyOS are split-phase. We expose these split-phase operations as TML "blocking" operations using our CPS transformation described above. For example, consider reading data from a sensor in TinyOS. The interface consists of a `getData` command and a `dataReady` event. In TML we expose a simple **sense** operation which we treat like a subcall. The handler is split, the code before the **sense** becoming the `getData` portion, and the code after becoming the `dataReady` portion.

Unfortunately, event-invoked continuations may occur at any time. Our scheduler depends on complete control over timing handlers. (Otherwise, the scheduler cannot promise meet deadlines for tokens scheduled with **timed_schedule** — the processor might be busy with an unanticipated action.) For this problem our work-around involves pushing the continuation code out of the TinyOS event handler (e.g. `dataReady`) and into its own, proper, token handler. The TinyOS event handler then **schedule**s the continuation token with highest priority.

However, a small (and predictable) amount of time is lost to the TinyOS event handler. Indeed, because TML is not a operating system and instead runs on top of TinyOS, it will always suffer time-leaks to event firings in the TinyOS subsystem. In reality, our current implementation performs only approximate timing. We make a hand-tuned estimate of the running time of token handlers based on only their TML-code, erring towards generous time allotments. If we underestimate the running time of an atomic action, there is a possibility that a **timed_schedule** will land slightly off. We believe that such approximate timing is good enough for most sensor networks applications. In future work we may attempt to improve timing by enabling abortion of atomic actions when they run over their time-allotments.

V. Example Applications of TML

In this section we will explain several example usages of TML. We will begin by adding a gradient interface to TML using a very simple program transformation, just as we did in our implementation of the subcall keyword. We will thus have built a small "gradient language" on top of core TML, one which provides a collection of gradient network coordination operations that mesh naturally with the token-oriented semantics of TML. This sort of lightweight language building is intended to represent typical usage for TML. After building up this gradient layer, we demonstrate its ease of use by writing a few simple applications. Finally, we discuss briefly how TML is used as a target language for the Regiment compiler.

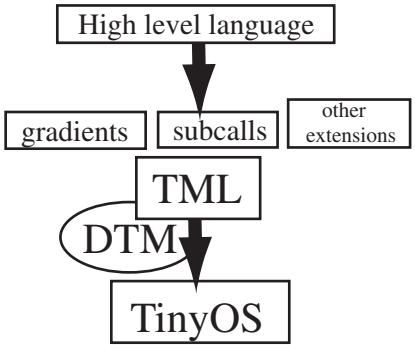

Fig. 2. The ingredients in the TML system. With Token Machines, the implementation process is bidirectional. We build TML *up* by enriching it with features like gradients, while compiling *down* from a higher level language. DTM is the underlying abstract model implemented by TML, and TinyOS provides the run-time environment.

A. Gradients

Gradients are a general purpose mechanism for breadth-first exploration from a source node. In simplest form, a gradient establishes a spanning tree that tells all nodes within the gradient how to route to the source as well as their hop-count. See Directed Diffusion [14] for an example of the utility of gradients and an overview of the design tradeoffs in gradient implementation.

Like our subcall facility, adding gradients involves a simple program transformation that adds code and implicitly appends extra arguments to token handlers. The extra arguments carry gradient information such as hop-count and version number. We use an interface consisting of four operations.

- **gemit**($\mathcal{T}_i, data \ldots$)
  gemit is a version of bcast which begins a gradient propogation from the current node. Gradient equality is determined by token name. Each node that joins the gradient will fire the $\mathcal{T}_i$ token handler. Subtokens are used to achieve overlapping, non-interfering gradients that use the same token handler.
- **grelay**($\mathcal{T}_i, data \ldots$)
  grelay is a version of bcast which continues the propogation of a gradient from the current node. **Grelay** fails silently if the named gradient has not been received.
- **greturn**($\mathcal{T}_{call}$, $\mathcal{T}_{via}$, $\mathcal{T}_{aggr}$, $data \ldots$)
  Greturn allows data to be propagated up gradients to their roots, with optional aggregation along the way. A **greturn** call sends data up the *via* gradient, and fires the *call* token handler on the data when it reaches the source. The *aggr* argument can be NULL, indicating no aggregation, or it can name a token handler of two arguments that can aggregate return values on their way to the source.
- **dist**($\mathcal{T}_i$)
- **version**($\mathcal{T}_i$)
  We do not expose parent pointers through the gradient interface, applications should not depend on the details of gradient implementation (e.g. single vs. multiple parents). But user code can use **dist** to become aware of its distance from the source

of a particular gradient. Further, if a gradient is re-emitted from a source node (as is often the case) the user code should be able to differentiate the different generations of gradient. To this end we allow the user to query the version of a gradient it has received. This is useful for performing initialization the first time a gradient is received. Both **dist** and **version** return -1 if the named gradient has not been received at the local node.

The above interface does not commit to a particular spanning tree selection or maintenance algorithm. The developer will want to choose a gradient implementation appropriate to the application. Ideally the gradient-augmented TML compiler should expose a set of choices of gradient implementation that covers the design space outlined in [14], but our current prototype compiler provides only simple, single-parent, link-quality unaware gradients.

B. Timed Data Gathering

This rudimentary example shows how to use the gradient interface to sample each node's light sensor . It uses a couple of simple keywords not mentioned above. The `startup` declaration indicates that the `Gather` and `GlobalTree` tokens will be scheduled when the node is first turned on. The `base_startup` keyword is similar, but only applies to the base-station node in the network. Also `BaseReceive` is predefined token handler supported only on the base-station, and used to return results to the outside world.

```
startup Gather, GlobalTree;
base_startup SparkGlobal;

token SparkGlobal() {
  gemit GlobalTree();
  timed_schedule SparkGlobal(10000);
}

token GlobalTree() {
  grelay GlobalTree();
}

token Gather() {
  greturn(BaseReceive,
          GlobalTree,
          NULL,
          subcall sense_light());
  timed_schedule Gather(1000);
}
```

This program emits a gradient from the base-station, which relays itself until it reaches the edge of the network, and refreshes itself every ten seconds. Once per second, every node fires the `Gather` token which uses the globally present gradient to route data back to the base-station.

C. Distributed Event Detection

Consider the problem of local event detection with unreliable sensors. We cannot trust the reading of a single sensor, but if several sensors within an area all detect an event, an alarm should be raised. Here we solve the problem by spreading out a small two-hop gradient from every node when it detects an event. When these gradients overlap sufficiently, the alarm is raised. This program assumes the declarations above, establishing the `GlobalTree`.

```
shared int total_activation;

token EventDetected () {
  emit AddActivation[MYID](1);
  schedule AddActivation[MYID](1);
}

token AddActivaton[sub] (int x) {
  if ( dist(self) < 2 )
    relay AddActivaton(x);
  total_activation += x;
  if (total_activation > threshold)
    greturn(BaseReceive, GlobalTree,
            NULL, ALARM);
  timed_call SubActivation[sub](1500, x);
}

token SubActivation[sub] (int x) {
  total_activation -= x;
  if (total_activation <= 0) {
    evict AddActivation[sub];
    evict SubActivation[sub];
  }
}
```

We keep the individual gradients from colliding by using subtokens for AddActivation and SubActivation (indexed by the ID of the node emitting the gradient). However, their overlap is still seen through the shared variable `total_activation`. This demonstrates the utility of lightweight gradients spawned and destroyed from arbitrary points in the network.

D. Leader Election

We will now build a reusable leader-election component in TML. All the nodes that invoke ElectLeader($\mathcal{T}_i$) will participate in the leader election for token $\mathcal{T}_i$. One such node will eventually be decided leader and receive an $\mathcal{T}_i$ token. Multiple leader elections can proceed concurrently in the network; this is because ElectLeader($\mathcal{T}_i$) uses subtokens indexed by $\mathcal{T}_i$ for all of its computation. The problem of garbage collecting dead tokens is ignored for the purpose of this example.

```
shared int winner;

token elect_leader(tokname T) {
  int current = winner;
  if (current == 0 || current < MYID) {
    winner = MYID;
    timed_schedule Confirm_Fire[T](5000, T);
    emit Compete(MYID, T);
  }
}

token Compete(int id, tokname T) {
  if (winner == 0) {
    winner = MYID;
    timed_schedule Confirm_Fire[T](5000, T);
  }
  if (version(Compete) == 0 || id > winner) {
    winner = id;
    relay Compete(id, T);
  }
}

token Confirm_Fire[sub](tokname T) {
  if (MYID == winner) schedule T();
}
```

E. Compiling Regiment

As a final application example we discuss our Regiment[17] compiler which targets TML. Regiment is a macroprogramming language in which a high-level program manipulates "regions" of sensor data as values in the language. Individual nodes in the network appear as data streams, and regions are groupings of these streams as designated by the programmer using a number of different criteria. The program operates over these streams and regions, performing

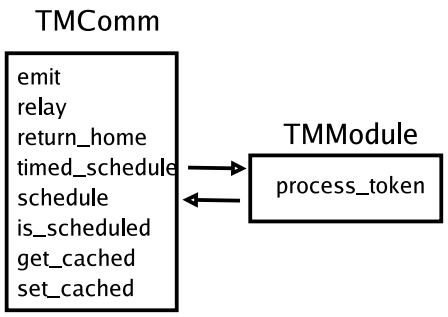

Fig. 3. The simple component structure of the TinyOS TML implementation.

actions which must be translated into node-level actions on local data. The resulting node behavior can be complex and difficult to reason about directly.

TML helps bridge this semantic gap by virtue of token-holders forming natural groups. Every expression in a Regiment program which evaluates to a region value gets assigned a *formation* token and a *membership* token. Every node that holds the membership token at a particular time is a member of region at that time. Formation tokens on the other hand, initiate the work of constructing or discovering the region. Formation tokens also have constraints on where and when they need to fire. Because of Regiment's clear, high-level semantics we can reason about where and when all of these events need to take place. Once this is done, TML makes it straight-forward to translate region-logic into token-logic.

VI. Structure of the TML Run-time

First we will examine the component structure of our Token Machine run-time. A complete TML program compiles to produce a single module conforming to the interface *TMModule*. This component provides only a single method, `process_token`, which takes a token and runs the appropriate handler. Each token handler becomes a single command in the generated code. The `process_token` message just looks at the type of the token dispatches appropriately. The rest of the system uses this `TMModule`, and handles the work of transporting tokens, the `TMModule` needs only to process tokens. This simple two component assembly is shown in Figure 3.

As described in Section V-A the gradient communication extensions can be built on top of core TML. But because gradients are an integral part of our use of TML, we choose to implement the gradient API directly in TinyOS. Hence you see commands `emit`, `relay`, and `return_home` listed in the `TMComm` interface. Most of the other methods appear exactly as they did in the discussion of TML's API.

However, the current TML implementation cuts one corner. Rather than each token handler having an arbitrarily structured private memory, in the current implementation each token handler stores exactly one cached copy of the corresponding token message arguments. Hence, `get_cached` and `set_cached` are the only access between a token handler and its stored memory. For example, if the user code needs information about parent pointers, hop counts, or version numbers it acquires it by using the `get_cached` method and extracting that information from the cached token. Stored fields

The shared memory, on the other hand, is allocated inside the `TMModule` component and need not appear in the interface between the two components.

A. BasicTMComm

The component used to provide the `TMComm` interface in the current TML implementation is called `BasicTMComm`. This component contains the scheduler and implements the communication interface. The scheduler maintains a queue of incoming tokens ordered in a way that respects timed tokens. Message reception events trigger event handlers which unpack the token from the message (currently one token per message), and place it in the queue. A separate task consumes messages from the queue, invokes `process_token`, and sets timers to schedule future consumptions.

`BasicTMComm` contains all the logic necessary to implement gradients. That is, none of it leaks into the `TMModule`. Emission is simply broadcast (plus book-keeping for counters and versions). Further, returning values via gradients requires that `BasicTMComm` route messages along the spanning trees. It does this by intercepting the message receives and never calling the `TMModule`'s `process_token` unless the return message has reached the root of the spanning tree.

VII. Evaluation and Discussion

TML is currently implemented as a high level simulator and as a compiler targeting the NesC/TinyOS environment. The mapping from Token Machines onto NesC was discussed in Section IV. Overall, it took relatively little effort to map TML into TinyOS because TML is not a *mechanism* so much as a *discipline*.

Our current compiler has some shortcomings with respect to the features laid out in this paper. Namely, the current implementation implements subcalls but makes all subcalls "direct" (as described in IV-A), and thus circumvents the necessity of the CPS transformation. As a result, we must manually insure that handlers complete in a relatively short time. This part of the implementation will be corrected in the near future.

Code size for compiled TML code is very good. Only a small constant factor size increase is added to the TML source when translated to NesC. The run-time support (DTM component) is also relatively lean. When compiled for the Mica2 mote, it consumes only 8836 bytes of ROM. RAM usage is worse: 817 bytes, including a token store of 320 bytes. Both RAM and CPU usage suffer as compared to "equivalent" native TinyOS code. This is because of the overhead of running the scheduler component and unnecessary copying of buffers. We believe that there are many optimizations yet to be exploited which can reduce memory redundancy. Future work will move in this direction.

What we have learned thus far from our use of TML is that its two important qualities are the atomic action model of concurrency, and the fact that communication is bound to persistent storage (tokens). The former precludes deadlocks and makes reasoning about timing extremely simple. The later essentially gives us a way to refer to communications that have happened through the token they leave behind. Also, tokens give us some of the benefits of "viral agent" type models of ad-hoc distributed computing (as seen in [5]), without the overhead. They can be seen as a lightweight version of this agent-oriented model.

Future Work and Conclusions

In the future, we will explore more dynamic alternatives for TML, including dynamic network retasking. TML programs are modular and conducive to division into "code capsules" similar to those employed by Maté. Ultimately, we concur with the position espoused in [22] that there should be separate representations for the end-user programming model, the code transport layer, and the execution

engine. Ideally, the user program should be compiled to a concise bytecode supported by a pre-installed virtual machine. We will look into targetting a virtual machine rather than native code, perhaps Maté itself.

We intend, however, to keep our focus on whole-program compilation. There are numerous optimizations (not described in this paper) currently under development that depend on whole program optimization. If we can determine all call-sites for a token handler, it is a great benefit. For example, unused arguments to token handlers can be eliminated (including automatically generated ones such as continuation and gradient arguments). Also the lack of pointers in TML results in a lot of unnecessary copying, some of which can be eliminated by a compiler that has access to the whole program.

We are also working on incorporating a token based routing scheme, allowing operations such as "route a Green token to all holders of a Red token". Eventually, we will also look at incorporating a larger array of established distributed algorithms in DTMs and TML. For example, we would like to incorporate quorums and consensus algorithms in a token-oriented manner.

REFERENCES

[1] Harold Abelson, Don Allen, Daniel Coore, Chris Hanson, George Homsy, Jr. Thomas F. Knight, Radhika Nagpal, Erik Rauch, Gerald Jay Sussman, and Ron Weiss. Amorphous computing. *Commun. ACM*, 43(5):74–82, 2000.

[2] R. Barr, J. Bicket, D. Dantas, B. Du, T. Kim, B. Zhou, and E. Sirer. the need for system-level support for ad hoc and sensor networks, 2002.

[3] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks, 2003.

[4] Cristian Borcea, Chalermek Intanagonwiwat, Porlin Kang, Ulrich Kremer, and Liviu Iftode. Spatial programming using smart messages: Design and implementation. In *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, March 2004.

[5] W. Butera. *Programming a Paintable Computer*. PhD thesis, MIT, 2002.

[6] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*, pages 164–175. ACM Press, 1991.

[7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *SIGPLAN Not.*, 39(4):502–514, 2004.

[8] Deepak Ganesan, Ben Greenstein, Denis Perelyubskiy, Deborah Estrin, and John Heidemann. An evaluation of multi-resolution search and storage in resource-constrained sensor networks. In *Proc. the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, November 2003.

[9] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80. ACM Press, 2004.

[10] Benjamin Greenstein, Deborah Estrin, Ramesh Govindan, Sylvia Ratnasamy, and Scott Shenker. DIFS: A distributed index for features in sensor networks. In *Proc. the First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[11] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 458–471. ACM Press, 1994.

[12] Wendi Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proc. the 5th ACM/IEEE Mobicom Conference*, August 1999.

[13] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104. ACM Press, 2000.

[14] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. International Conference on Mobile Computing and Networking*, August 2000.

[15] Philip Levis and David Culler. Mate: a tiny virtual machine for sensor networks. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 2002.

[16] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TAG: A Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. the 5th OSDI*, December 2002.

[17] Ryan Newton and Matt Welsh. Region streams: Functional macroprogramming for sensor networks. In *Proceedings of the First International Workshop on Data Management for Sensor Networks (DMSN)*, August 2004.

[18] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings 1992 ACM Conf. on Lisp and Functional Programming, San Francisco, CA, USA, 22–24 June 1992*, pages 288–298. ACM Press, New York, 1992.

[19] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrating communication and computation. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 430–440. ACM Press, 1998.

[20] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proc. the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI '04)*, March 2004.

[21] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. the International Conference on Mobile Systems, Applications, and Services (MOBISYS '04)*, June 2004.

[22] Bridging The Gap: Programming Sensor Networks with Application Specific Virtual Machines. In submission to osdi 2004.

[23] Yong Yao and J. E. Gehrke. The Cougar approach to in-network query processing in sensor networks. *ACM Sigmod Record*, 31(3), September 2002.