FiGaRo: Fine-Grained Software Reconfiguration for Wireless Sensor Networks

Luca Mottola[1], Gian Pietro Picco[2], and Adil Amjad Sheikh[2]

[1] Department of Electronics and Information,
Politecnico di Milano, Italy
mottola@elet.polimi.it

[2] Department of Information and Communication Technology,
University of Trento, Italy
{amjad,picco}@dit.unitn.it

Abstract. Wireless Sensor Networks (WSNs) are increasingly being proposed in scenarios whose requirements cannot be fully predicted, or where the system functionality must adapt to changing conditions. In these scenarios, the ability to reconfigure *portions* of the software running on WSN nodes becomes imperative. At the same time, recent WSN proposals often employ *heterogeneous* nodes (e.g., sensors and actuators), which require the deployment of different code on different devices, based on their characteristics. Unfortunately, existing work in the field largely focuses on simpler scenarios where the same, monolithic program is distributed to all the nodes in the WSN.

In this paper we present FIGARO, a programming model supported by an efficient run-time system and distributed protocols, collectively enabling an unprecedented fine-grained control over *what* is being reconfigured, and *where*. Using FIGARO, the programmer can deal explicitly with component dependencies and version constraints, as well as select precisely the subset of nodes targeted by reconfiguration, leaving the others unaltered. We show that our run-time support imposes a very limited processing and memory overhead, while the communication overhead lies within 9% of the theoretical optimum.

1 Introduction

The nodes of a wireless sensor network (WSN) are often deployed in large numbers and inaccessible places, making individual code uploading an impractical solution. This problem was early recognized in the WSN research field, leading to solutions exploiting the wireless link for on-the-fly, untethered software reconfiguration [1]. However, these solutions were designed to suit the needs of early WSN architectures, i.e., application-specific systems with homogeneous devices.

Problem and Motivation. Today, WSNs are proposed in contexts where their functionality changes over time and/or cannot be predicted a priori. For instance, in emergency response [2] systems the WSN must be reconfigured on-the-fly by mobile operators which demand customized behavior to carry out their activities. In similar scenarios, anticipating all expected needs, if at all possible, may lead to complex and unreliable code

R. Verdone (Ed.): EWSN 2008, LNCS 4913, pp. 286–304, 2008.

cluttered with rarely-used functionality. Therefore, software reconfiguration—even if representing a rare activity compared to the application operations—becomes a much-needed feature. For reconfiguration to be fully effective, however, programmers must retain fine-grained control over *what* is being reconfigured, by updating selected functionality to minimize energy consumption. However, most platforms allow updates only of the full application image. In the very few exceptions, programmers sorely miss proper constructs to deal with dependencies among different functionality, versions, and other fundamental aspects of reconfiguration [1].

Moreover, modern WSNs are typically heterogeneous, containing a mixture of sensing devices and/or actuators. In building monitoring, for instance, a wide range of sensor and actuators is deployed, e.g., to implement heating, ventilation, and air conditioning (HVAC) control [3]. As different nodes are likely to run different application code, software reconfiguration may be limited to a specified portion of the WSN. For instance, a structural engineer inspecting a building may want to load a new piece of functionality only on seismic sensors deployed in a specific location (e.g., the floor being inspected), to process the sensed data in a previously unanticipated manner [4]. In this case, fine-grained control over *where* the code is deployed, based on application attributes of the nodes, is largely missing from existing approaches, which instead are designed to distribute the *same* code to *all* the nodes, regardless of their function [1].

Contribution. In this paper we present FIGARO (FIne Grained softwAre RecOnfiguration), a novel approach enabling fine-grained control over *what* is reconfigured and *where*, to a degree unprecedented in WSNs. FIGARO tackles the two problems in an integrated way, spanning all the aspects from the programming model down to the node-level run-time support and the protocols for efficient code distribution. Its programming model, described in Section 2, has two core constituents:

- the *component model* defines constructs for structuring the code on the single nodes. Differently from other component models for WSNs (e.g., [5]), ours is designed with reconfiguration in mind, thus providing dedicated constructs to deal with component dependencies and versions, and to simplify the reconfiguration process.
- the *distribution model* defines constructs to restrict component dissemination only to a given subset of nodes—the reconfiguration target—based on programmer-specified characteristics of the nodes or their current software configuration.

Our implementation includes a lightweight node-level run-time system, discussed in Section 3, whose responsibility is to manage the local part of a reconfiguration process, along with an efficient protocol for code distribution, illustrated in Section 4. Both are evaluated in Section 5. As for the former, our results show that processing and memory overhead are almost negligible, while the energy overhead during reconfiguration is marginal. Similarly, our distributed protocol results in a communication overhead within 9% of the theoretical optimum, which is instead computed in a centralized manner and with global knowledge of the system topology.

In Section 6 we compare FIGARO against representative state-of-the-art systems. Finally, in Section 7 we conclude by illustrating directions for future work.

2 Programming Model

FiGaRo is currently built on top of the Contiki [6] operating system, and therefore relies on the C programming language.

2.1 Specifying *What* Is Reconfigured

Components, Interfaces, and Dependencies. In FiGaRo, a *component* represents a single unit of functionality and deployment. The services provided by a component are described by its *interface*. For instance, Figure 1 shows the declaration of an interface for data collection. This specifies the signature of two operations to broadcast interests and to report the data, respectively. Components must provide the code for all the operations in the interface declaration, as in the case of Figure 2. The **DECLARE_COMPONENT** macro is used to specify the name of the component (`tree_routing`), the interface it implements (`data_collection_if`), and the component version (2).

To accomplish its goal, a component normally interacts with others on the same node. Interaction occurs through function calls across components using **CALL**, as shown in the first operation of the component in Figure 2. However, it is not for granted that a component provides an (interface containing the) operation required by another, while the caller component may not be able to continue its execution without a callee component implementing the required interface. Therefore, the presence of a **CALL** statement determines a *dependency* between caller and callee.

In FiGaRo, dependencies are explicitly declared by the programmer using the **DECLARE_DEPENDENCY** macro. The first parameter of this macro is a *receptacle*, the dual of an interface. An interface specifies a set of operations provided by a component to others, while a receptacle specifies the set of interfaces a component requires

```
DECLARE_INTERFACE(data_collection_if, {
  void (* broadcast_interest)(void* data,  u8_t len);
  void (* report)(uip_ipaddr_t dest, void* data,  u8_t len); })
```

Fig. 1. An example of component interface

```
DECLARE_COMPONENT(tree_routing, data_collection_if, 2)
DECLARE_DEPENDENCY(radio_receptacle, radio_if, 3, MANDATORY | STATIC)
void broadcast_interest(void* data,  u8_t len) {
  CALL(radio_receptacle, send(&broadcast_addr, &msg, 64));
  // ...
}
void report(uip_ipaddr_t dest, void* data,  u8_t len) {
  // ...
}
ON_RUNNING({ // ON_SUSPEND, ON_DESTROY are also available
  // ...
})
```

Fig. 2. A component implementing the interface of Figure 1

from others. In the case of Figure 2 the dependency being declared specifies the name of the receptacle (radio_receptacle), the interface required (radio_if), and the minimum component version allowed for a component (3). Moreover, the programmer can also specify a bit-masked constant describing the *nature* of the dependency. In the example, **MANDATORY** specifies that the component cannot run without relying on the needed interface. Otherwise, the dependency is considered optional, and the component is expected to work correctly also in absence of the specified interface. Instead, **STATIC** indicates that once a callee component is bound to the caller through the receptacle, the callee component cannot be changed. Otherwise, a reconfiguration can take place substituting the component with another providing the same interface.

Figure 3 shows an example of component configuration. The Sampling component is responsible for querying the sensor, and calling the report function in TreeRouting, which transmits the data to a sink. Note how TreeRouting satisfies only the **MANDATORY** dependency of Sampling, while the **OPTIONAL** one is currently not satisfied. This information is reflected in the receptacle descriptor inside the run-time support, as described in Section 3.

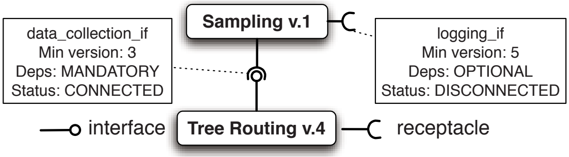


Fig. 3. An example of component configuration

Component Life Cycle. The life cycle of a component is illustrated in Figure 4. A component becomes **RUNNING** when all its dependencies on other components are satisfied, i.e., components implementing the required interfaces are available on the node. Note that dependencies are inherently recursive, i.e., a component may depend on some others, which in turn may depend on others, and so on. Therefore, the instantiation of a component may trigger the instantiation of an entire component *closure*, based on the declared dependencies. In practice, however, WSN applications are made of a small number of components with short dependency chains. The instantiation of a set of components bound by dependencies occurs atomically, i.e., control returns to the application only when the instantiation of *all* components is complete. When a component providing services to others undergoes a reconfiguration, the components exploiting those services move to the **SUSPENDED** state, and revert to the **RUNNING** state when the reconfiguration completes. Instead, the **DESTROYED** state is reached when the component has been replaced by another with the same interface.

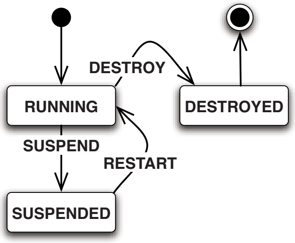


Fig. 4. The life cycle of a FiGaRo component

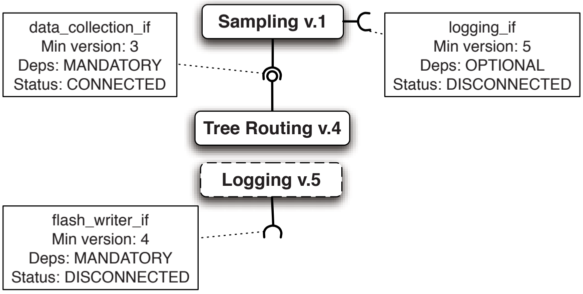
Programmers can intervene at each step of the life cycle by specifying code fragments to be executed when entering a given state, as shown in Figure 2. When starting a new component, for instance, the body of the **ON_RUNNING** macro is executed. Similar operations exist for each state. The ability to intercept run-time activities is particularly important in the case of **SUSPEND**, to give programmers the ability to release resources held by the suspended components, and avoid deadlocks and run-time faults.

Component Reconfiguration. In FIGARO, programmers do not need to manage the reconfiguration manually, e.g., using a dedicated API as in [7]. Instead, the underlying run-time automatically and transparently manages the reconfiguration process, based on dependencies and component versions. When components are instantiated at start-up, the run-time keeps track of their version, the interface they implement, and their dependencies. Upon receipt of a new component C, reconfiguration unfolds as follows. Provided C's **MANDATORY** dependencies can be satisfied:

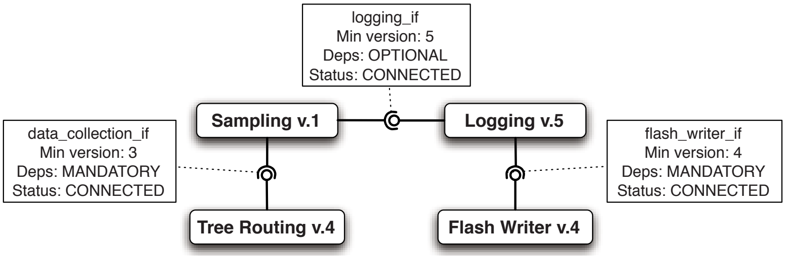
1. C is instantiated if there is no running component with the same interface, or
2. C replaces another component C_{old} implementing the same interface as C if:
   (a) C's version is greater than C_{old}'s,
   (b) no component currently relying on C_{old} has a **STATIC** dependency on it.

If a component cannot be instantiated because of one or more unsatisfied **MANDATORY** dependencies, it is buffered in the hope that the necessary components will be received later on. If this does not happen, the component is discarded after a timeout.

As an example, Figure 5 shows a possible evolution of the configuration shown in Figure 3. When a `Logging` component is received, the node-level run-time determines that it can be used to satisfy the optional dependency of `Sampling`. However,



(a) `Logging` is received.



(b) `FlashWriter` is received.

Fig. 5. A sample evolution of the component configuration in Figure 3

Logging has a **MANDATORY** dependency of its own, which cannot be satisfied. Therefore, Logging is temporarily buffered and remains disconnected from the other components, yielding the configuration in Figure 5(a). In Figure 5(b), a FlashWriter component satisfying the dependency of Logging is received. The run-time determines, by recursively travelling the component graph, that all dependencies are now satisfied, and instantiates the new components in the correct order (i.e., FlashWriter before Logging), yielding the configuration shown in the figure.

Our automatic reconfiguration mechanism relieves the programmer from checking the conditions for the reconfiguration to take place, changing the component interconnections, and managing the coordination among the components involved. Although similar approaches (e.g., [8]) already proved their effectiveness in other contexts, to the best of our knowledge we are the first to enable this functionality in WSNs.

2.2 Specifying *Where* Reconfiguration Occurs

FiGaRo empowers programmers with the ability to delimit the portion of the WSN where reconfiguration takes place. This is achieved with dedicated programming constructs that enable programmers to: *i)* declare the attributes characterizing a node; *ii)* specify the reconfiguration target—i.e., the subset of nodes for component deployment—by using boolean predicates over the nodes' attributes.

```
DECLARE_NODE({
  Function = SENSOR
  Type = TEMPERATURE
  Floor = 1
  Battery = getBatteryReading()
})
```

Fig. 6. Declaring node attributes

```
DECLARE_TARGET({
  Function == SENSOR && Battery >= 70 &&
  (Type == TEMPERATURE || Type == VIBRATION) &&
  RUNNING(TreeRouting) &&
  VERSION(TreeRouting) <= 11
})
```

Fig. 7. Declaring the reconfiguration target

Figure 6 shows an example where we use the **DECLARE_NODE** macro to specify that a node hosts a temperature sensor and is located on a given floor. Note how, in principle, attributes can be assigned any legal C expression, including C functions as in the case of the Battery field. The nodes targeted by the reconfiguration can be specified declaratively as an (arbitrary) boolean predicate over node attributes using the macro **DECLARE_TARGET**. In Figure 7, we specify as reconfiguration target the set of temperature or vibration sensors with at least 70% of battery left, and running a TreeRouting component with version less than 11. Notably, the latter requirement leverages off information automatically exported by our run-time layer, which describe the current component configuration on a node. Specifically, the parametric, built-in predicate **RUNNING** takes as input the name of a given component C, and yields true when evaluated on a node where C is currently in such state. Instead, the built-in function **VERSION** returns the version of the component given as parameter.

3 Node-Level Run-Time Support

FIGARO provides the constructs described in Section 2.1, concerned with node-level reconfiguration, by making extensive use of C macros, therefore moving at compilation time most of the added complexity while not requiring any dedicated pre-processing step. However, dynamic reconfiguration requires specialized run-time support, provided by library functions we developed, linked against the (unmodified) Contiki kernel.

Our run-time maps FIGARO components to Contiki services [6], and leverages off Contiki's dynamic linking facility [9] to install new code. Consequently, the implementation of the **CALL** macro uses Contiki look-up functions to find a pointer to the callee component, and perform the operation requested. Interfaces and receptacles are represented by descriptors (standard C structs) containing an array of function pointers. In the case of interfaces, these always point to the corresponding functions in the component currently implementing the interface. Instead, the pointers inside receptacles are assigned the function pointer values of the associated interface, when connected, or NULL otherwise. In addition, receptacle descriptors contain further fields to keep track of the nature of dependency, as well as the minimum version required by any component connected to it, as shown in Figure 5.

Based on the information gathered by our macros during the compilation phase, our run-time maintains on every node an internal representation of the exported attributes and current software configuration. This is represented as a graph where vertexes are components, and edges are labeled to reflect the nature of the dependency at hand, similarly to Figure 5. When a new component arrives, simple graph traversal algorithms are used to check the conditions for the installation of a new piece of functionality. If the new component can indeed be installed, the run-time fires the relevant state transitions on all involved components, installs the new component by reconfiguring the involved receptacles, and updates the graph accordingly.

Instead, the constructs concerned with the reconfiguration target, illustrated in Section 2.2, require a minimal amount of pre-processing. On the user base-station, reconfiguration is triggered using a dedicated executable, whose arguments are two files: one containing the component binary image and one with the reconfiguration target (e.g., as in Figure 7). A dedicated pre-processor we developed parses them together, generates a unique reconfiguration identifier, divides the binary image into smaller chunks fitting in single physical messages, and starts injecting them into the network. The details of the routing protocol determining their propagation are described next.

4 A Routing Protocol for Selective Code Distribution

Our dedicated distribution scheme revolves around two base mechanisms:

- While the application is running, we exploit its message traffic to build a *mesh* topology interconnecting all nodes with same attribute-value pairs, as in Figure 8, to identify all possible alternative paths connecting the relevant nodes.
- When a reconfiguration is requested, a subset of the mesh paths are exploited to build a tree rooted at the target node closest to the injection point, as in Figure 9. The tree is then used to propagate the component chunks to all target nodes.

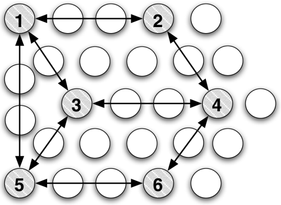


Fig. 8. A mesh connecting all target nodes

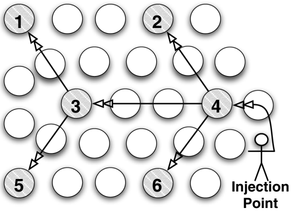


Fig. 9. A distribution tree exploiting the mesh

In principle, the two mechanisms above could be designed independently. Nonetheless, our solution is explicitly conceived to take advantage of their mutual interplay. As our objective is to build shortest paths to the target nodes, we make all paths in the mesh itself *bi-directional*. This allows us to exploit the same shortest paths regardless of where the code is injected. Moreover, our solution is designed to create a *planar* mesh topology, i.e., one in which no two paths with different end-points cross at any intermediate node, as in Figure 8. Results in graph theory indeed demonstrated how planar graphs involve fewer routing loops [10]. As a result, the tree topology built atop the mesh easily identifies near-optimal paths, as we demonstrate in Section 5.

4.1 Building the Mesh Topology

Architecture and Data Structures. As the mesh is built during normal system operation, we must minimize the impact of the mesh-building protocol on the application behavior. We obtain this goal by designing a solution that does *not* generate explicit control messages. Rather, we leverage off the application traffic by piggybacking the current value of a node's attributes on every outgoing message[1]. This is achieved by interposing a thin software layer between the application and the underlying network layers whose interface is the same as the original network stack, making its use transparent to the application.

The information piggybacked is overheard by all nodes in range[2], and used to populate a simple routing routing table (e.g., as in Figure 10), that describes the paths of the mesh. Each entry in the table contains a node identifier and the associated attribute-value pair, the next hop to reach that node along with the corresponding cost in hops, and a timestamp to discriminate stale information. In addition, the Bridging and Bridge Cost fields are used to distinguish entries corresponding to bidirectional paths. The former possibly contains the identifier of another node with same attribute-value pair, representing the opposite end-point of the path itself, whereas the latter stores the total path length in hops. Each entry in the table is associated with a lease (not shown) that, if not refreshed, causes the entry removal.

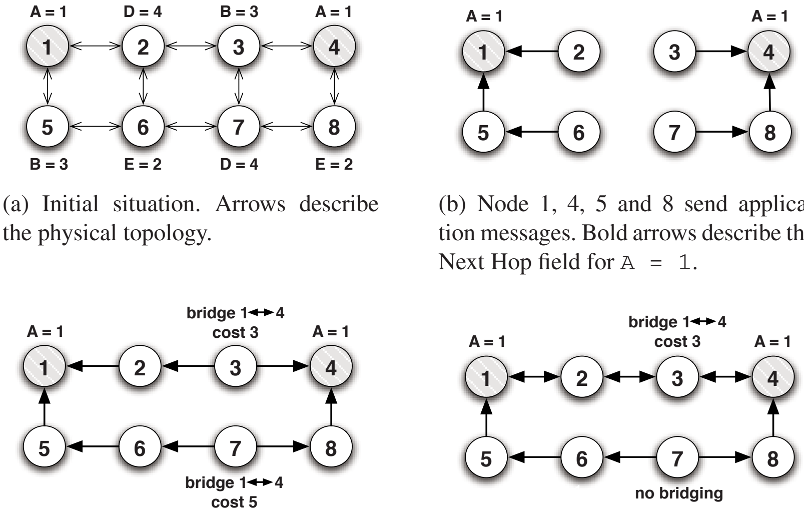
Protocol Operation. Figure 11 describes an example of mesh construction. The initial situation, depicted in Figure 11(a), illustrates the physical network topology and the attributes defined in the node declarations, along with their corresponding values. Initially, all routing tables contain only entries relative to the local node. For instance, let us focus on the nodes having attribute `A` equal to `1` as target. When node 1 first

[1] In case a node is silent, we generate dummy messages at a pre-specified rate.

[2] A simple hook within the Contiki radio layers allows us to overhear also unicast messages.

Source	Attribute	Value	Cost	Bridging	Bridge Cost	Next Hop	Timestamp
Node 3	B	3	0	null	null	self	4
Node 4	A	1	1	Node 1	3	Node 4	25
Node 1	A	1	2	Node 4	3	Node 2	72

Fig. 10. Routing table at node 3 in the situation of Figure 11(c)



(a) Initial situation. Arrows describe the physical topology.

(b) Node 1, 4, 5 and 8 send application messages. Bold arrows describe the Next Hop field for A = 1.

(c) Node 3 and node 7 recognize a chance to build a bidirectional path connecting node 1 and node 4.

(d) The path through node 3 is complete. The one through node 7 is pruned as unnecessary.

Fig. 11. Example of mesh construction (grey circles are target nodes)

sends an application message, we append a subset of node 1's routing table entries to it[3]. The nodes in range parse this information, increments all cost fields by one, and add these entries to their routing tables provided no other entry with same attribute-value pair but smaller or equal cost exists. By doing this at every node, node 1's specification spreads across multiple hops. For instance, node 5's piggybacked information also includes node 1's initial entry, as it was overheard from node 1's transmissions. Assuming node 4 and 8 eventually send some application message as well, the resulting situation is as depicted in Figure 11(b).

To recognize when a bidirectional path can be established, we look for received entries containing an attribute already stored in the local table, but from a different source and greater or equal cost. This is the case in Figure 11(c), where node 3 receives from node 2 an entry for attribute A with value 1 and cost 2. In this situation, a bidirectional path for the same attribute can be established, with node 1 and node 4 as end-points. To establish the bidirectional path in both directions, we insert the newly received entry in

[3] Entries are selected in round-robin, their number limited by a configuration parameter.

node 3's routing table with the Bridging field set to the identifier of the opposite end-point of the path (e.g., node 4 in case of node 3 in the last entry of Figure 10), and the Bridge Cost field set to the total cost of the path itself. Similarly, we update any entry already in the table that refers to the other end-point of the bidirectional path—as it is the case for the second entry in Figure 10—modifying the Bridging and Bridge Cost accordingly. Afterwards, entries with non-null Bridging fields are propagated only towards the node reported in the Bridging field itself. Thus, the second entry in Figure 10 is propagated only towards node 1, whereas the last entry spreads only towards node 4. This is as simple as appending an optional field to all outgoing messages stating what nodes propagate what entries.

As a side-effect of the above processing, more than a single bidirectional path connecting node 1 and node 4 could be established. For instance, a further path is eventually built through node 5, 6, 7 and 8, with a total cost of 5. This, however, poses unnecessary communication overhead. To alleviate this undesirable behavior, non-null Bridging entries are propagated only if the node is not aware of other (bidirectional) paths with smaller cost. In our example, node 7 eventually stops propagating its non-null Bridging entry after overhearing the last entry at node 3, which contains a smaller cost. This ultimately yields the situation in Figure 11(d). Although this scheme does not completely prune all redundant paths, it greatly diminishes their number. Pruning all the paths but the shortest one would indeed require propagating the minimum cost entry multiple hops away from the shortest path. How far to propagate is hard to determine without knowledge of the network topology. Also, the additional paths may be used as back-ups in case of sudden faults. We plan to investigate this in the near future.

Dynamic Attributes and Topology Changes. The protocol operation occurs whenever the application generates network traffic. Therefore, in the case of time-varying attributes, the accuracy provided by the mesh topology w.r.t. the current values of attributes is ultimately dictated by the amount of application traffic over time. Applications generating more traffic allow our protocol to build more accurate topologies, whereas it is difficult to do so if the amount of traffic flowing in the network is insufficient to keep up with the dynamics of the varying attribute. As for topology changes, e.g., due to failing nodes, invalid routes will eventually expire without being refreshed. As soon as the application generates further messages, our protocol identifies alternative routes according to the new topology. Still, the time taken to build the new routes is dependent on the amount of traffic generated by the application.

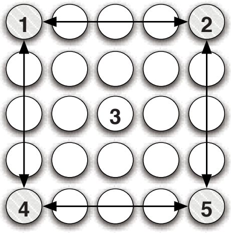


Fig. 12. Node 3 has equal cost to all target nodes

Enforcing Planarity. By construction, our scheme does not generate multiple paths with different end-points crossing at an intermediate node. Indeed, the only way this can be obtained is to have, in the same routing table, more than one non-null Bridging entry for the same attribute-value pair with different source. Consider Figure 12: node 3 may try to establish two crossing paths, e.g., connecting node 1 to node 5 and node 2 to node 4. This cannot occur in our protocol, as received entries with cost greater or equal to the local table for the same attribute-value pair are ignored, end every non-null Bridging entry can be used to establish a single bidirectional path. Therefore, node 3 in Figure 12 will never be able to generate crossing paths.

4.2 Distributing Code

When a reconfiguration takes place, code is distributed along a tree: redundant paths in the mesh are identified based on the position of the code injection point, using a *marker* message. This contains the reconfiguration identifier generated by our pre-processor, and an encoding of the predicate defining the required reconfiguration target. The former serves to support multiple concurrent reconfigurations. The latter is used by nodes to determine, based on their routing table, the next hop for the marker. Upon forwarding, target nodes add to the marker the cost accumulated along the last bidirectional path traversed. This way, the marker eventually reaches all the target nodes, making them aware of their distance from the injection point. This information is used at each target node to configure a dedicated distribution tree by selecting as parent the target node that, along the links of the mesh, is the closest to the injection point. The selection is communicated to the parent with a message containing the identifiers of the source target node and of the selected parent. Note how code dissemination can start before the entire tree is built. When receiving a code chunk, a node that has not yet determined its children simply defers forwarding and buffers the chunk. Buffering would happen in any case, since a component cannot be reconstructed until all chunks are received.

The code distribution phase demands reliable communication, e.g., because all code chunks must be correctly delivered. We employ a simple hop-by-hop reliability mechanism, based on implicit acks. Nodes on a tree path buffer every message, waiting for the downstream node to re-send it. When this occurs, the upstream node overhears the transmission, and concludes the message was received; otherwise, it is re-sent. Similar techniques have already been successfully employed in WSNs [11]. However, our implementation decouples this aspect, enabling the use of alternative reliability schemes.

5 Evaluation

To assess the effectiveness of our approach, in this section we separately evaluate the performance of the node-level run-time support, and of the code distribution protocol.

5.1 Evaluating the Node-Level Run-Time Support

Our objective here is to quantify the overhead imposed by FIGARO w.r.t. plain Contiki. We consider the following performance figures:

- The *memory occupation* caused by our component model, w.r.t. both program and data memory. We evaluated the former by looking at the size of binary images after compilation. As for the latter, we manually inspected the code managing components and their interconnections, looking for any data structure we defined.
- The additional *processing time* caused by the presence of components. This is affected both by the installation of a new component compared to the native Contiki

dynamic linker, and by function calls across components using **CALL** instead of a direct C call. As for the latter, we placed the call in a loop and repeated the operation a million times, since the single call is too quick to be measured precisely.

- The *energy consumption* during reconfiguration, which may increase as a result of the additional processing required to manage components and dependencies.

We measured processing time and energy consumption using real nodes as opposed to simulation environments, as similar fine-grained aspects are only partially modeled in existing simulators. Practically, we measured the processing overhead using a JTAG programmer attached to the node to measure the time elapsed between the execution of different instructions. Energy consumption was instead evaluated using an Agilent 54832B oscilloscope and a multimeter hooked to a node, which in our case was a TMote Sky [12]. We repeated the experiments concerning these metrics 5 times using 3 different nodes, and averaged the results. New components have been injected via a USB cable attached to the node, to avoid any bias due to the radio.

To gather the above metrics, we employed a `Blinker` component offering a single interface with two operations to start/stop the blinking of a led. We varied the number of receptacles within the component itself to evaluate our performance w.r.t. a varying number of dependencies. The processing within `Blinker` is the same as in [9], and is quite simple being described by only 17 lines of C code. This choice was intentional, as simpler components make the overhead more evident w.r.t. the above metrics.

Results. Figure 13 shows the memory overhead, which turns out to be quite reasonable, w.r.t. both program and data memory. As for the former, the binary code deployed in addition to the operating system accounts for less than 2 Kbytes in total. This cost, along with the overhead due to helper data structures, is paid once and for all, regardless of the number of components and the number of their interfaces/receptacles. Conversely,

Performance Measure	Memory	Footprint
Dependency Checks	Program	1.1 KB
Helper Functions	Program	802 bytes
Helper Data Structures	Data	230 bytes
Per-Component Data	Data	15 bytes
Per-Interface Data	Data	8 bytes
Per-Receptacle Data	Data	10 bytes

Fig. 13. Memory overhead

Function Type	Time Overhead %
Empty	157.5%
50 integer additions	20.1%
3 x 3 matrix inversion	5.4%
5 x 5 matrix inversion	0.98%
Fourier Transform (100 input values)	0.78%
Fourier Transform (1000 input values)	0.03%

Fig. 14. FiGaRo calls across components vs. native C function calls

Dependencies	Time (s)		Energy (mJ)	
	Absolute	Overhead	Absolute	Overhead
1	0.518 sec	+0.019	3.45	+0.07
2	0.520 sec	+0.021	3.45	+0.07
3	0.525 sec	+0.026	3.47	+0.09
4	0.528 sec	+0.029	3.49	+0.11
5	0.532 sec	+0.033	3.5	+0.12

Fig. 15. Time and energy to install the Blinker component

the bottom section of Figure 13 reports the memory consumption incurred every time a component, interface, or receptacle is loaded on a node. In this case as well, the overhead is fairly limited. Based on these results, we maintain that our approach can scale to a sizable number of components simultaneously running on the same node, presumably well beyond the current needs of common WSN applications. As for the amount of code to be deployed, we compared the size of the binary image of the plain-Contiki Blinker process used in [9] against ours, implemented as a FiGaRo component. The size increases from 1.01 Kbytes to 1.11 Kbytes, yielding an overhead of only 9.98%. We believe this value is good, given the little complexity of the processing at hand.

The overhead in performing calls across components against direct C function calls is reported in Figure 14. Interestingly, when the function called does not contain any real processing the overhead due to using **CALL** is high. In this case, performing the look-up of the Contiki service implementing the requested component dominates the processing time. In contrast, some even simple processing within the function called makes this metric drop abruptly. For instance, in the case of a Fourier transform (e.g., employed to perform in-network processing in WSN applications such as [4]) the overhead becomes less than 1%. Therefore, although our programming model does introduce an overhead, the performance penalty is expected to be negligible in real applications.

By the same token, the time for installing a new component, and hence the energy consumed during this process, increases only marginally w.r.t. the standard Contiki dynamic linker, as shown in Figure 15 for a varying number of dependencies in the component being installed. Note how these values are independent of the size of the component being deployed, as they represent the overhead imposed by our run-time layer in addition to the Contiki dynamic linker, which we left unmodified. Also, they scale well with the number of dependencies, showing only a very small increase. To place Figure 15 in context, consider that the energy *overhead* in the case with 5 dependencies is equal to only about 5% of the *total* energy required to transmit a 32-byte message.

5.2 Evaluating the Code Distribution

In this section we assess the effectiveness of our solution for code distribution by reporting about simulations performed using Cooja, the Contiki simulator.

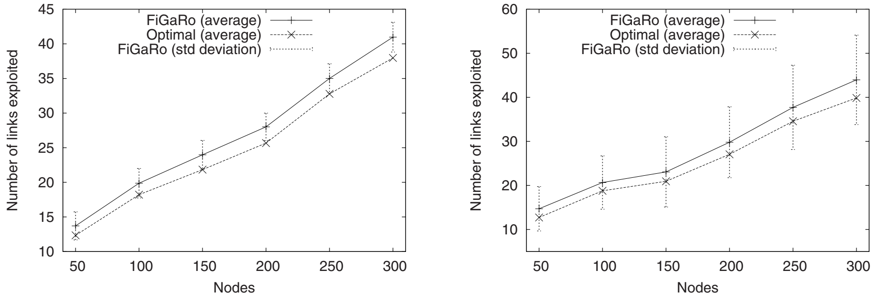
The evaluation of code distribution protocols for WSNs has hitherto focused on metrics such as latency and message overhead [1]. However, these are usually affected by mechanisms other than the distribution protocol itself. For instance, latency is affected also by the MAC layer protocol, as back-off timers, random transmission delays, and transmission slots in TDMA schemes are employed to reduce collisions. Similarly, message overhead is affected by the specific reliability mechanism employed.

However, the above concerns are orthogonal w.r.t. the problem we are tackling and the *essence* of the solution we presented, whose performance is determined *primarily* by the shape of the tree used during the distribution phase. Indeed, the number of hops separating the injection point from the target nodes strongly impacts both latency and message overhead irrespective of the MAC layer and reliability mechanism employed, which instead affect the individual 1-hop transmissions. Therefore, we chose to evaluate our protocol by focusing on the *number of links* employed during the code distribution phase[4], and compared this metric against the optimal distribution tree computed with a shortest path algorithm and global knowledge of the network topology. We also measured the *convergence speed* of our mesh-building algorithm, i.e., how many messages the application must generate for the routing tables to stabilize. In both cases, we rely on the standard Contiki MAC layer as implemented in Cooja. Moreover, we used the reliability mechanism discussed in Section 4, for which simulations confirm a 100% delivery in all the experiments discussed next.

As for simulation settings, each node exports a single attribute whose value is randomly selected at start-up. Reconfiguration targets are defined by a single equality predicate on this attribute. During the mesh-building phase each node sends an application message every $5 + \mathcal{D}$ seconds. $\mathcal{D}$ is a random delay we introduced to avoid locking effects among nodes, and to generate executions with varying traffic rates at different nodes. Application messages are 64 bytes in size, to which we piggyback 24 bytes of control information corresponding to 4 entries from the local routing table. During the simulations, the mesh-building phase takes place first. The convergence speed is determined when routing tables at all nodes do not change for 5 consecutive message sends. At this point, the mesh is considered stable: a random node is chosen as injection point and the tree-building phase is started. We discuss results obtained in regular grids and random topologies. In the former, each node can communicate with 4 neighbors. This setting models some of the applications we target (e.g., indoor WSN deployments [13]). In the latter the number of neighbors varies from 3 to 7. For each scenario, we averaged the results over 20 repetitions with varying distribution scopes and injection points.

Results. Figure 16 shows how the number of links exploited by our solution varies according to the system size and topology. Remarkably, the performance of our protocol remains always within 9% of the theoretical optimum, and is almost constant as the number of nodes increases. By examining the simulation logs, we realized that the gap is mostly due to cases where it may be more convenient to access the mesh from more than a single entry point. When this does not hold and the injection point is very close to a target node (i.e., within 2-3 hops), the average gap w.r.t. the optimal solution is even lower, around 3%. This confirms that our mesh-building algorithm, thanks to

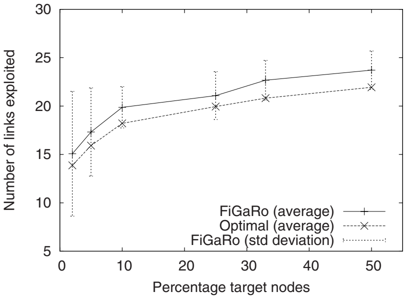
[4] In cases where nodes can forward a message towards n neighbors with a single physical packet we still count n links, as most reliability mechanisms would send separate messages anyway.



(a) Grid topologies

(b) Random topologies

Fig. 16. FIGARO performance vs. topology and system size (target nodes are 10% of the total)



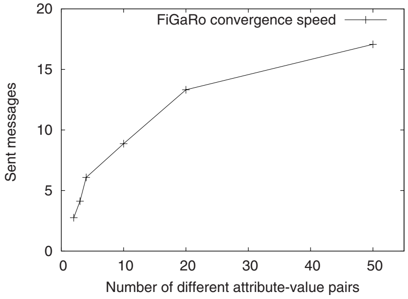


Fig. 17. FIGARO performance vs. number of target nodes (100 nodes arranged in a grid)

Fig. 18. FIGARO convergence speed (100 nodes arranged in a grid)

its planarity property, yields near-optimal routes in the distribution trees relying on it. Further, note how Figures 16(a) and 16(b) exhibit similar trends, although the results on random networks show higher variability due to the irregularity of the topology.

Figure 17 provides a different perspective by analyzing the behavior of our protocol w.r.t. the percentage of target nodes. As shown in the chart, our solution is barely affected by this parameter. The high variability observed with few target nodes is due to cases where nodes end up aligned w.r.t. the injection point, and the distribution tree degenerates in a chain. In these configurations, intermediate nodes are reached at essentially no cost. The probability of these configurations decreases as the number of target nodes grows. We limited our experiments to half of the nodes in the system as targets. Beyond this point, the scenario starts bearing similarities with traditional code distribution in homogeneous networks, where all nodes are target. In this case, existing solutions are better suited, e.g., [14].

Finally, we verified that the convergence speed of the mesh-building phase is not affected by the system scale. Indeed, the extent to which routing entries are propagated is not dictated by the overall number of nodes, rather by the amount of redundancy among attribute-value pairs. This claim is supported by Figure 18, showing the number of messages required to build the mesh against the number of (distinct) attribute-value pairs in the system. When the latter is small the mesh builds quickly, as the bidirectional paths connecting nodes with the same attribute-value pairs are likely to be short. Instead, when attribute-value pairs are highly heterogeneous the mesh takes more time, due to the dual argument. Overall, the values in the chart are good: only 17 messages need to be sent when 50 different attribute-value pairs are present, i.e., only 2 nodes in the 100-node network of Figure 18 have the same attribute-value pair—a rather unusual setting. In any case, the values in the chart should represent only a very little fraction of the overall system lifetime, typically measured over months or even years.

6 Related Work

Single-Node Reconfiguration. Several solutions enable the installation of new code on individual nodes. At the operating system level, besides Contiki also the SOS operating system [15] provides dynamic linking, while FlexCup [16] enables this functionality in TinyOS [17], where this was initially not possible. These solutions concentrate on efficient dynamic linking, and are therefore complementary to our approach. In principle, our component model can be re-applied in SOS and FlexCup with minimal modifications, as it is mostly based on standard C macros. We chose Contiki because, unlike FlexCup, it preserves the application state as it does not require a reboot after code loading and, in comparison to SOS, its service functionality eases the implementation of the FiGaRo component model. Alternative approaches use interpreted languages and virtual machines (e.g., [18,19,20]), with some also allowing for extensible instruction sets, e.g, [21]. Nonetheless, the trade-offs between interpreting code and executing native binaries, as discussed in [18], suggest the use of the latter for long-running systems where reconfiguration is a rare event, as in the scenarios we target.

Most importantly, none of the above approaches provides support to the *programmer* for managing the interactions among the different functionality on a node during reconfiguration. Indeed, even though component models for WSN programming have already been proposed (e.g., [2, 5, 7]), they do not include any dedicated construct for managing mutable component configurations. Conversely, we made component dependencies and versions first-class citizens in the FiGaRo programming model, and designed the reconfiguration mechanism by balancing automation and customizability.

Code Distribution. To the best of our knowledge, we are the first to provide efficient distribution of code to an arbitrary subset of nodes identified by programmer-provided information. Our distribution model is inspired by Logical Neighborhoods [22], a programming abstraction giving developers the ability to define system partitions based on application information. A message-passing API is then provided to interact with nodes in a given partition. Although [22] describes a generic communication layer for Logical Neighborhoods, tackling the issues germane to code distribution required a completely different routing support, as described in Section 4. In the field of code distribution, the approach closest to ours is the TinyCubus framework [23], where code can be distributed to all nodes with a given role, e.g., all cluster-heads. This is far less flexible than FiGaRo's predicate logic over programmer-defined attributes, and does not encompass the ability to identify the target nodes based on their current software configuration, e.g., as provided by the **RUNNING** built-in-predicate. At the network level, TinyCubus assumes *a priori* knowledge of the system topology and of the location of nodes with a given role, as it requires to specify an upper bound on the number of hops separating nodes with the same role. In contrast, our solution is fully dynamic and decentralized.

Network-wide distribution of code has been widely investigated, tackling different facets of the problem. On one hand, solutions have been proposed to reduce the size of the code to be distributed by employing differential patching and smart linking mechanisms, e.g., [24, 25]. Still, similar concerns are orthogonal to the problem we tackle in this work, and the corresponding solutions may be integrated in our framework for even better performance, e.g., by injecting a patch instead of the whole binary when the new component is going to replace an older version. Instead, other approaches focused on routing. Trickle [14] uses a counter-based technique called "polite gossip", whose objective is to suppress redundant transmissions while guaranteeing eventual delivery. Deluge [26] uses a similar technique, with the addition of a negotiation phase to guarantee the proper sequencing of packets. This is also used in MNP [27] to address the hidden terminal problem before transmitting the actual code. Sprinkler [28] and Firecracker [29] instead leverage off node hierarchies, by first sending code to "core" nodes up in the hierarchy, which then forward the code to nodes in their vicinity. As the objective of all the above solutions is to distributed code to all nodes, they can avoid any background activity under normal operating conditions. For the same reason, however, these mechanisms are hardly applicable in our case. For instance, it would be fairly inefficient to add multi-hop negotiation in Deluge to address the case where the target nodes are multiple hops away.

7 Conclusion and Future Work

In this paper we presented FiGaRo, a solution enabling software reconfiguration in WSNs at an unprecedented level of granularity, both w.r.t. the functionality to reconfigure on single nodes, and the subset of nodes targeted by the reconfiguration. We provide a component-based programming model with explicit support for component dependencies and versions, along with a dedicated component life cycle, and an intuitive yet expressive distribution model allowing programmers to identify what part of the network is affected by the reconfiguration. Our evaluation demonstrated how the overhead imposed on single nodes is negligible, while the communication overhead during reconfiguration lies within 9% from the theoretical optimum.

Our research agenda includes distributed mechanisms to provide more guarantees (e.g., atomicity) w.r.t. the reconfiguration process. For instance, the programmer may require that either all or none of the nodes in the reconfiguration target install the new component, to tolerate run-time faults where a node crashes and then reboots.

Acknowledgements. The authors wish to thank Fabio Fabbri for helping with the measures gathered on real nodes, Prof. Greg Frederickson for the insightful discussions on the shortest path problem on planar graphs, and Alessandro Ungari for his work on the implementation of the FiGaRo run-time. The work described here was partially supported by the European Union under the IST-004536 RUNES project.

References

1. Wang, Q., Zhu, Y., Cheng, L.: Reprogramming wireless sensor networks: challenges and approaches. IEEE Network 20(3) (2006)
2. Costa, P., et al.: The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In: PERCOM. Proc. of the 5^{th} Int. Conf. on Pervasive Communications (2007)
3. Deshpande, A., Guestrin, C., Madden, S.: Resource-aware wireless sensor-actuator networks. IEEE Data Engineering 28(1) (2005)
4. Lynch, J.P., Loh, K.J.: A summary review of wireless sensors and sensor networks for structural health monitoring. Shock and Vibration Digest (March 2006)
5. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: A holistic approach to networked embedded systems. In: PLDI. Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation (2003)
6. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Proc. of 1^{st} Wkshp. on Embedded Networked Sensors (2004)
7. Grace, P., Coulson, G., Blair, G., Porter, B., Hughes, D.: Dynamic reconfiguration in sensor middleware. In: MidSens. Proc. of Int. Wkshp. on Middleware for Sensor Network (2006)
8. Becker, C., Handte, M., Schiele, G., Rothermel, K.: PCOM - A component system for pervasive computing. In: PERCOM. Proc. of the 2^{nd} Int. Conf. on Pervasive Computing and Communications (2004)
9. Dunkels, A., Finne, N., Eriksson, J., Voigt, T.: Run-time dynamic linking for reprogramming wireless sensor networks. In: SenSys. Proc. of 4^{th} Int. Conf. on Embedded Networked Sensor Systems (2006)
10. Frederickson, G.: Fast algorithms for shortest paths in planar graphs, with applications. Siam J. Computing 16(6) (1987)
11. Stann, F., Hiedemann, J.: RMST: reliable data transport in sensor networks. In: Proc. of the 1^{st} Int. Wkshp. on Sensor Network Protocols and Applications (2003)
12. MoteIV Technology, `www.moteiv.com`
13. Stoleru, R., Stankovic, J.: Probability grid: A location estimation scheme for wireless sensor networks. In: SECON. Proc. of the 1^{st} Int. Conf. on Sensor and Ad-Hoc Communication and Networks (2004)
14. Levis, P., Patel, N., Culler, D., Shenker, S.: Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In: NSDI. Proc. of the 1^{st} Conf. on Networked Systems Design and Implementation (2004)
15. Han, C.C., Kumar, R., Shea, R., Kohler, E., Srivastava, M.: A dynamic operating system for sensor nodes. In: MobiSys. Proc. of the 3^{rd} Int. Conf. on Mobile Systems, Applications, and Services (2005)
16. Marrón, P.J., Gauger, M., Lachenmann, A., Minder, D., Saukh, O., Rothermel, K.: FlexCup: A flexible and efficient code update mechanism for sensor networks. In: Römer, K., Karl, H., Mattern, F. (eds.) EWSN 2006. LNCS, vol. 3868, pp. 212–227. Springer, Heidelberg (2006)
17. Hill, J., et al.: System architecture directions for networked sensors. In: Proc. of the 9^{nt} Int. Conf. on Architectural Support for Programming Languages and Operating Systems (2000)
18. Levis, P., Culler, D.: Maté: a tiny virtual machine for sensor networks. In: ASPLOS-X: Proc. of the 10^{th} Int. Conf. on Architectural Support for Programming Languages and Operating Systems (2002)
19. Müller, R., Alonso, G., Kossmann, D.: A virtual machine for sensor networks. In: Proc. of the EuroSys Conf. (2007)
20. Koshy, J., Pandey, R.: VM*: synthesizing scalable runtime environments for sensor networks. In: SenSys. Proc. of 3^{th} Int. Conf. on Embedded Networked Sensor Systems (2005)

21. Levis, P., Gay, D., Culler, D.: Active sensor networks. In: NSDI. Proc. of the 2^{nd} Conf. on Networked Systems Design & Implementation (2005)
22. Mottola, L., Picco, G.P.: Logical Neighborhoods: A programming abstraction for wireless sensor networks. In: DCOSS. Proc. of the the 2^{nd} Int. Conf. on Distributed Computing on Sensor Systems (2006)
23. Marrón, P.J., Lachenmann, A., Minder, D., Hahner, J., Sauter, R., Rothermel, K.: Tinycubus: a flexible and adaptive framework sensor networks. In: Proc. of the 2^{nd} European Workshop on Wireless Sensor Networks (EWSN) (2005)
24. Reijers, N., Langendoen, K.: Efficient code distribution in wireless sensor networks. In: WSNA. Proc. of the 2^{nd} Int. Conf. on Wireless Sensor Networks and Applications (2003)
25. Koshy, J., Pandey, R.: Remote incremental linking for energy-efficient reprogramming of sensor networks. In: Proc. of 2^{nd} European Workshop on Wireless Sensor Networks (EWSN) (2005)
26. Hui, J.W., Culler, D.: The dynamic behavior of a data dissemination protocol for network programming at scale. In: SenSys. Proc. of 2^{nd} Int. Conf. on Embedded Networked Sensor Systems (2004)
27. Kulkarni, S., Wang, L.: MNP: Multihop network reprogramming service for sensor networks. In: ICDCS. Proc. of the 25^{th} Int. Conf. on Distributed Computing Systems (2005)
28. Naik, V., Arora, A., Sinha, P., Zhang, H.: Sprinkler: A reliable and energy efficient data dissemination service for wireless embedded devices. In: RTSS. Proc. of the 26^{th} International Real-Time Systems Symposium (2005)
29. Levis, P., Culler, D.: The Firecracker protocol. In: Proc. of the 11^{th} ACM SIGOPS European Workshop (2004)