Managing Software Evolution in Large-scale Wireless Sensor and Actuator Networks

BARRY PORTER, GEOFF COULSON and UTZ RÖDIG, School of Computing and Communications, Lancaster University, UK

Wireless Sensor and Actuator Networks (WSANs) will increasingly require support for *managed software evolution*: i.e., systematic, ongoing, efficient and non-disruptive means of updating the software running on the nodes of a WSAN. While aspects of this requirement have been examined in the literature, the big picture remains largely untouched, resulting in the generally static WSAN deployments we see today. In this paper we propose a comprehensive approach to managed software evolution. Our approach has the following key features: (i) it supports *divergent evolution* of the WSAN's software, such that different nodes can evolve along different lines (e.g. to meet the needs of different stakeholders, or to address localised adaptations) and (ii) it supports both *instructed and autonomous evolution* such that nodes can be instructed to change their software configuration or can evolve their own configuration (e.g. to manage rapidly-changing environmental conditions where remote micro-management would be infeasible due to the high latency of the WSAN environment). We present the four intra-WSAN protocols that comprise our solution, along with an accompanying server-side infrastructure, and evaluate our approach at scale.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution and Maintenance

General Terms: Management, Algorithms

Additional Key Words and Phrases: Wireless Sensor Networks, Software, Evolution, Components

1. INTRODUCTION

Wireless Sensor and Actuator Network (WSAN) deployments are becoming increasingly large-scale, multi-purpose and 'infrastructural' in nature. Pertinent examples include smart buildings, campuses and cities [Bernat 2010; various 2011] or multi-functioned environmental management systems [Smith et al. 2009]. Furthermore, because of their multi-purpose nature and the major capital investments involved, we envisage that the owners/operators of future infrastructural WSAN deployments will increasingly be driven to adopt an agile, multi-stakeholder, usage model. Such deployments will need to run divergent and ever-changing software configurations across their (heterogeneous) node base.

These considerations point to an increasing need for *managed software evolution* in sensor networks – i.e. systematic, ongoing, efficient and non-disruptive means of updating the software running on the nodes of a WSAN. For example, WSANs will need software updates to fix errors, to adapt to varying environmental conditions, and to manage performance problems (e.g. optimising network performance by migrating cluster heads or stream processing modules) [Hughes et al. 2008; Cao and Stankovic 2008]. Furthermore, in a multi-stakeholder environment, different stakeholders using a facility over its lifetime will typically need *different* software as they come and go, and the software requirements of individual stakeholders will vary over time [Steffan et al. 2005; Huygens et al. 2010]. This is all needed within the context of networks that are inherently characterised by low-throughput, high-latency, high-loss and intermittent communication along with highly limited resources in general – characteristics that are compounded over the wide multi-hop dispersals in which future WSANs are expected to operate.

From our analysis of the above trends and characteristics we derive the following key requirements for a comprehensive managed software evolution approach for WSANs:

— **Divergent evolution**. We define this as the capability to evolve the software configuration of a given node independently from that of its neighbours. It is increasingly the case that different nodes in a WSAN perform different functions and run different code. Examples including different nodes sensing different aspects of their environment as needs dictate, or different nodes performing different networking duties. Given this, it is increasingly inappropriate for software update mechanisms to optimise for the increasingly-rare case that updates will be applied uniformly to all nodes (as is done in classic work such as [Hui and Culler 2004]). Instead, we must provide efficient support for the general case of divergent evolution.

— **Instructed and Autonomous evolution**. *Instructed* software evolution is that specified directly by administrators in the form 'install component X on node Y'; while we define *autonomous* evolution as the ability of the software on each node to reason about itself and make changes to its configuration without consulting a central manager. Given the large-scale, high-latency, low-bandwidth, and volatile nature of infrastructural WSANs, it is increasingly the case that decisions to change a node's software configuration must be made on the basis of local, real-time, observations. Examples include deciding to switch to a more appropriate routing protocol, or to become a cluster head, or to migrate a stream processing function, all of which are based on locally-available information at sensor nodes. Given this, it is increasingly impractical to adopt the exclusively instructed (i.e. centralised, micro-managed) evolution approach assumed by classic work. Instead, we must additionally support autonomous evolution and harmonize this with instructed evolution to avoid conflicts.

As well as these two core requirements we identify the following additional constraints: (i) *Minimality*: a well-designed software evolution approach should avoid reliance on specialised infrastructure such as out-of-band network paths or complex network-level protocols, instead building directly on unreliable one-hop network capabilities. This reflects the highly constrained resources of typical WSAN nodes which often cannot afford more complex protocols such as IP or AODV. (ii) *Unobtrusiveness*: The primary reason to deploy a WSAN is to gather data on aspects of the real world; a value-added system should therefore operate in the background as much as possible with minimal impact on the sensor/actuator traffic. (iii) *Energy Efficiency*: WSANs are often energy constrained and as such our solution should be minimal in computation and (especially) inter-node communication. Indeed, minimising energy consumption tends to be much more important than minimising the latency of software updates.

This paper proposes a middleware approach to managed software evolution that addresses all of these requirements and constraints. The middleware can be viewed as a virtual 'meta-WSAN' that 'senses' software configurations within the WSAN and performs 'actuation' on those configurations. In more detail, our approach employs a lightweight suite of protocols which collectively tracks nodes currently in the WSAN; reports their software configurations; performs software dissemination and instructed configuration updates on behalf of users (e.g. installation of new code); and harmonizes these updates with autonomously-initiated configuration changes.

Administrators interact with our middleware using GUI-based clients that present a simple view of the network and its current software configuration, as well as allowing that configuration to be easily changed by dropping new modules onto selected nodes.

The remainder of this paper is organised as follows: in Section 2 we survey related work, demonstrating that little research has comprehensively addressed the issue of managed software evolution in infrastructural WSANs. In Section 3 we provide essential background on our Lorien OS which we use as the implementation platform for our protocols; then in Section 4 we present our approach in detail; and in Section 7 we evaluate it. We conclude and offer an outlook in Section 8.

2. RELATED WORK

To the best of our knowledge, no previous work has proposed a comprehensive, scalable, network-wide, solution for managed software evolution in infrastructural WSANs. In particular, all WSAN-based software updating designs of which we are aware assume that updates are applied uniformly (no divergent evolution) and that individual nodes cannot change their own software configuration (instructed evolution but no autonomous evolution). More flexible software evolution approaches exist in the world of PC-class systems (e.g. Microsoft's Active Directory) but these are clearly inapplicable to resource-scarce environments like WSANs.

Nevertheless, there does exist a significant and growing body of research on software updating for WSANs. We distinguish two main categories of work in this area: i) image-based approaches and ii) modular approaches.

Image-based approaches (which are mostly based on TinyOS [Hill et al. 2000]) have received the vast majority of researchers' attention for software updates. They divide into an *entire-image* sub-category, in which a full system image is disseminated to all nodes in the WSAN [Hui and Culler 2004]; and a *differential-image* sub-category, in which an attempt is made to conserve network bandwidth by disseminating 'diffs' that are applied offline at the target node to generate a new system image [Reijers and Langendoen 2003; Panta et al. 2011]. Unfortunately, neither of these sub-categories support either divergent or autonomous evolution and, furthermore, it is not easy to see how they could be extended in the future to support these key properties:

— *Divergent evolution*. For entire-image approaches, divergent evolution is inherently problematic because, short of deploying an expensive routing infrastructure, the only way to send a new image to a subset of nodes is to flood it to all nodes [Hui and Culler 2004] (with consequent high overhead). Furthermore, divergent evolution is problematic for differential-image approaches because diffs are *context dependent*: each will only work on a specific and pre-determined software configuration. This means that the more the WSAN's software diverges, the closer we approach a pathological extremity in which each node needs its own dedicated diffs, even when a required change is uniform across the board (e.g. updating a common module m on every node).
— *Autonomous evolution*. For image-based approaches instructed software evolution is the norm but *autonomous* evolution is inherently problematic: for entire-image approaches a node needs to keep in its secondary storage (e.g. external flash memory) a distinct complete image to underpin every conceivable autonomous change it might want to enact, leading either to a severe limit on the number of possible changes or an extremely high storage overhead. Furthermore, autonomous evolution is problematic for differential-image approaches because diffs cannot be built and stored at nodes in advance (again, because of context dependency), and the process of building a diff locally is too computationally heavy for typical WSAN nodes.

Modular approaches[1] (e.g. [Han et al. 2005; Cao et al. 2008; Dunkels et al. 2004]) are potentially more promising. These differ from image-based approaches in working on the basis of smaller code units called 'modules' which have the key properties of *fine granularity* and *independence of context*. This means that they can be both disseminated and instantiated with much greater efficiency than image-based approaches. Furthermore, they can be cached in a node's secondary storage from where they can be loaded when required and flexibly composed with the rest of a software system to generate a very large number of valid configurations.

[1]For our purposes, *virtual machine* approaches (e.g. [Brouwers et al. 2009; Levis and Culler 2002]) essentially share the same characteristics as modular approaches.

However, despite this potential to offer new capabilities and open up radical new avenues of WSAN systems research, no present-day modular system of which we aware offers any actual support for divergent or autonomous evolution. In fact, the state of the art is such that only LiteOS [Cao et al. 2008], SOS [Han et al. 2005], Contiki [Dunkels et al. 2004] and our own Lorien OS offer any support for even basic updating (e.g. Contiki only supports one updatable module at any one time loaded on top of a fixed kernel). A wider point is that no present-day modular system, apart from Lorien, has a basic architectural reflection capability, which we see as a pre-requisite for the support of autonomous change (see next section). In this paper we present what we believe to be the first complete network-scale management solution for modular software approaches in low-power WSNs which satisfies the requirements of both divergence and autonomy to which modular approaches are in principle so well suited.

3. BACKGROUND ON LORIEN

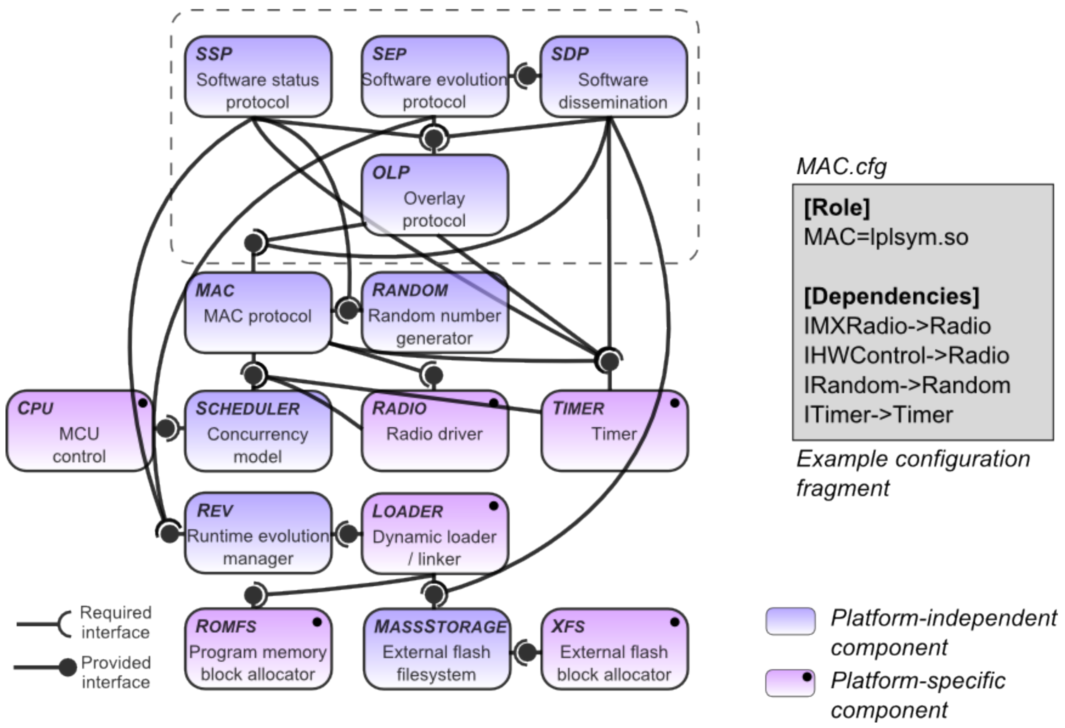
Lorien [Porter and Coulson 2009; Porter et al. 2011] is a modular WSAN OS that supports software updating in terms of the runtime addition, removal and replacement of software modules (referred to in Lorien-parlance as ‘components’). It is assumed that components to be added to the running system are already present in the host node’s secondary storage, having been placed there previously by external network protocols such as those presented in this paper. Lorien type-checks updates to help assure system integrity across change.

Lorien is notable for the *generality* of its software updating approach. In particular, OS-level functionality in Lorien is itself built from components that can be added/removed/replaced just like any application component. This goes beyond kernel-based WSAN OSs such as Contiki [Dunkels et al. 2004] or SOS [Han et al. 2005] which only support the updating of higher-level components. In fact, Lorien offers no prescribed ‘kernel’ at all: a Lorien system is simply a set of components whose composition provides the functionality currently required by the user. For instance, the example configuration shown in Fig. 1 includes:

— generic OS functionality (SCHEDULER, TIMER);
— basic communication support (RADIO and MAC);
— dynamic loading support (REV, LOADER, ROMFS and MASSSTORAGE)[2];
— the software evolution management protocols that are central to this paper (i.e. OLP, SSP, SEP and SDP; see Section 4).

A further feature of Lorien is its support for *architectural reflection*: Lorien maintains on each node a run-time representation of that node’s current software configuration, on the basis of which the node can autonomously reason about changes it may wish to make. In more detail, a Lorien configuration is described by a node-resident set of *configuration fragments*, each of which specifies the abstract name of a single component instance (‘role’) within the overall configuration, and the component type from which that instance is sourced, along with all of that role’s dependencies (see the example configuration fragment on the right hand side of Fig. 1). The set of configuration fragments describing a node’s current software configuration is held in a so-called *manifest* held in persistent memory. Further configuration fragments (and the component types to which they refer) are held in a node’s secondary storage (e.g. external flash memory) and software evolution is effected simply by adding/removing configuration fragments to/from the manifest from secondary storage: this directly causes the addition/removal of corresponding components to/from the running system. All of

[2]Note that even these four components can be replaced or unloaded just like any other component (although removing them precludes any further software evolution).



```
[Role]
MAC=lplsym.so

[Dependencies]
IMXRadio->Radio
IHWControl->Radio
IRandom->Random
ITimer->Timer
```

Fig. 1. An example Lorien node configuration. – Components interact via strongly-typed provided/required interfaces. Each component instance (e.g. SSP, SEP etc) is associated with a specific object file and described by a 'configuration fragment'. The fragment declares a unique abstract 'role' name for this component instance, the component type from which to source this instance, and describes how to connect the required interfaces (dependencies) of this instance to other abstractly-named instances in the system. Dependency specification for an instance *X* is of the form *RequiredInterfaceN->Y*, indicating that the specified **required** interface of *X* should be connected to the type-compatible **provided** interface of the component instance abstractly named *Y*. The protocol suite described in this paper is shown within the dotted rectangle.

this behaviour is encapsulated within the REV component in Fig. 1 which exports a collection of 'software evolution commands' (see [Porter et al. 2011]).

Using these facilities any software component running on a Lorien node is free to inspect the current software configuration in terms of the instantiated roles and their types and dependencies. This facilitates autonomy by allowing components acting as 'autonomy agents' to experiment with instantiating roles from different component types (within given constraints such as their being dependency-compatible) and indeed by adding and removing roles in general – perhaps from a pool of optional roles – and to observe the effects of such changes in terms of relevant performance metrics.

Finally, it is not necessary in Lorien for a component to have all of its dependencies satisfied at all times: for example, in the context of Fig. 1 it is possible to instantiate MAC without RADIO being present. In such cases Lorien's REV component ensures that such a component is not connected to others (e.g. SDP) that depend on it (and so on recursively). When a component is later instantiated that satisfies some of these hanging dependencies, the appropriate connections are automatically made, and any components that now have all of their dependencies satisfied immediately 'spring into life' (returning to our example, SDP would spring into life as soon as MAC had its RADIO dependency satisfied). This mechanism assures system integrity across change while avoiding the need for complex quiescence mechanisms [Pissias and Coulson 2008]. It also has the side effect that the order in which components are added to/removed from a node is irrelevant. This has important implications for our managed software evolution system as discussed in Section 5.2. Further detail on this and other aspects of Lorien is available in the literature [Porter and Coulson 2009; Porter et al. 2011].

4. DESIGN OVERVIEW

We now present our central contribution: a comprehensive, scalable, network-wide solution for managed software evolution in WSANs. While this design was originally developed on top of Lorien, it is in principle applicable to any modular OS environment with reflective capabilities.

Outside the WSAN our design employs two architectural elements: *portals* and *clients*. A portal is a program that runs on a PC-class machine connected to one or more 'gateway' nodes of the WSAN (see Fig. 2). Portals interact with the WSAN using the protocol suite discussed below, and provide a 'portal API' that exports web service operations enabling remote management of the software deployed in the WSAN. Clients are potentially-remote programs that use the portal API to mediate between human users and the WSAN. Various client programs can be used – such as GUI-based interfaces or clients that provide scripted batch-based execution of commands.

Inside the WSAN we require a suite of four networking abstractions that support the operations offered by the portal API. These abstractions assume as the networking infrastructure only a basic MAC layer offering 1-hop unreliable unicast and broadcast of small fixed-size frames (110 bytes in the case of Lorien on the TelosB platform). They are defined as follows:

— **Multi-channel staged collection.** A primitive that unreliably delivers one frame of data one hop closer to a sink node with support for multiple such 'channels'. At each hop the data frame is delivered to a locally registered handler for that channel which may process that frame before recursively using this primitive for the next hop.
— **Multi-fragment information delivery.** A primitive that builds on the above to reliably deliver a large data collection from a given node to a sink node such that the sink node can re-constitute that information into its original whole.
— **Multi-channel node instruction.** A primitive that allows the portal to unreliably send instructions to selected nodes, again with multi-channel capacity, such that the locally registered node handler for a channel can take action on that instruction.
— **Multi-fragment information dissemination.** A primitive that allows the portal to reliably propagate a large data item (such as a file) in a fragmented fashion towards selected nodes and notify those nodes that the data item has arrived.

When working in the context of a larger system these networking abstractions may be supported by, or built upon, communication protocols that are shared by other subsystems and applications. For our purposes here, however, we specify these protocols in detail as standalone entities. Our implementing protocol suite is carefully designed to minimise the amount of traffic it generates and the amount of state it maintains at each node, and to defer reliability/retransmission issues to the portal/client level in accordance with the end-to-end principle. In addition, the suite is designed in a modular way so it can be straightforwardly adapted to different environments – e.g. for environments that support a routing infrastructure, our basic overlay protocol OLP could be replaced with a protocol such as AODV or OLSR.

The standalone protocol suite that we use to underpin the above communication abstractions for managed software evolution comprises the following four protocols:

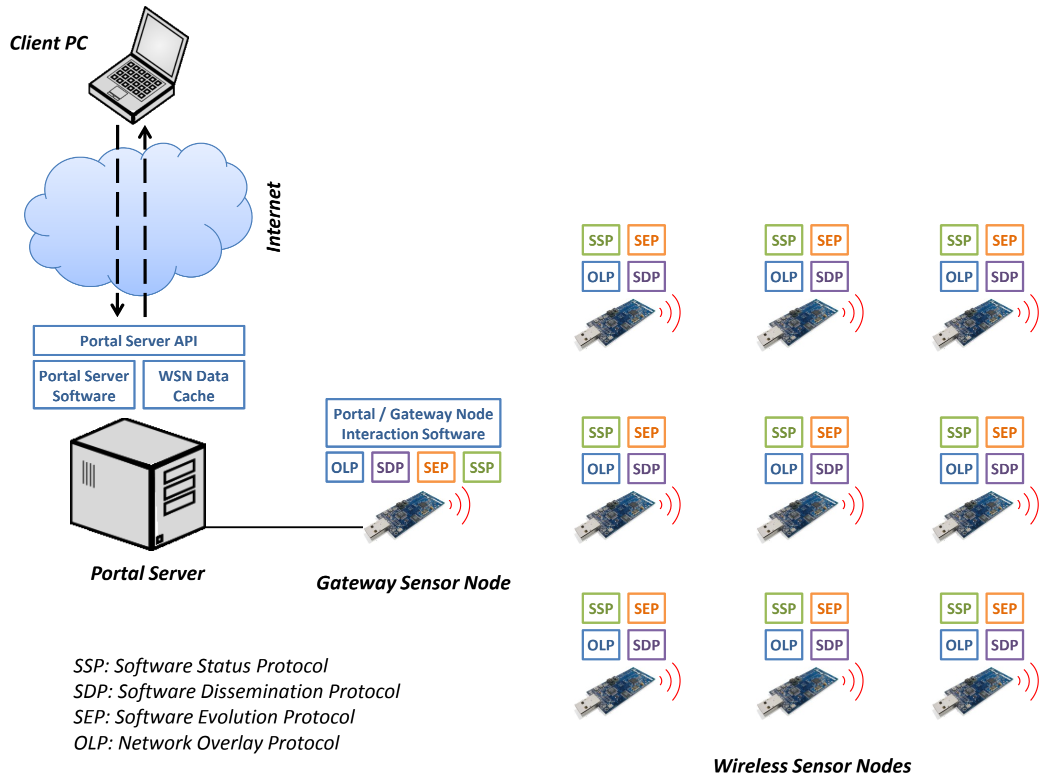

Client PC
Internet
Portal Server API
Portal Server Software
WSN Data Cache
Portal Server
Portal / Gateway Node Interaction Software
OLP
SDP
SEP
SSP
Gateway Sensor Node
SSP: Software Status Protocol
SDP: Software Dissemination Protocol
SEP: Software Evolution Protocol
OLP: Network Overlay Protocol
Wireless Sensor Nodes

Fig. 2. Overview of our design. We discuss all elements shown in detail with the exception of client software. Sensor node software executes on the Lorien modular OS.

— **Overlay protocol.** OLP supports the above limited form of unreliable communication between nodes and the portal and carries SSP, SEP and SDP data both to and from the portal via a simple hop-by-hop primitive.
— **Software status protocol.** SSP uses OLP to report the presence of nodes to the portal, together with the detailed current software configuration of each node, using a per-hop buffering strategy to rate-limit its traffic to a low constant. SSP on selected nodes can additionally be signalled by the portal to recover missing packets in the set of packets that describe a given node's overall software configuration.
— **Software evolution protocol.** SEP uses OLP to carry Lorien's software evolution commands (add/remove/replace) from the portal to selected nodes that have been instructed to evolve along a new path.
— **Software dissemination protocol.** SDP propagates new software modules to selected nodes when requested to do so by SEP. It uses OLP to assist in the directed propagation of module files to help efficiently support divergent software evolution.

Figure 2 illustrates a typical instantiation of our overall design. The portal runs on a PC connected to a 'gateway' node in the WSAN that communicates wirelessly with other nodes. Each node runs a Lorien instantiation including components that encapsulate our OLP, SSP, SEP and SDP implementations. The design is entirely plug-and-play, holding all necessary information within the network itself: When the portal is started it begins to receive, via SSP, soft-state information that describes the nodes in the network and the (possibly divergent) software executing on them. When a new node comes within range of any existing node, or when an existing node autonomously changes its software configuration, SSP immediately reports that information to the portal. This kind of server-less operation, in which all pertinent data is known to the network itself rather than held centrally on a server, is an important factor in addressing our goal of autonomous software evolution harmonised with instructed evolution.

In the following sections we first describe in detail the ‘infrastructure elements’ (i.e. portals and clients) and then discuss each of the above four protocols in turn.

5. INFRASTRUCTURE

5.1. Portals

The role of the portal is to mediate between clients and the OLP / SSP / SEP / SDP protocols in the WSAN. Internally, the portal maintains a *portal cache* that reflects its current view of the population of WSAN nodes and their software configurations.

The portal API, exported by the portal and used by clients, is as follows:

— **int update(String clientEndpoint; Cmd c; ID node; Comp C$_{old}$[], C$_{new}$[])** initiates changes to the software configurations of the specified node. *Cmd* can be *add* to add new components (C$_{old}$ is null in this case), *remove* to remove components (C$_{new}$ is null), or *replace* (i.e. replace C$_{old}[i]$ with C$_{new}[i]$). *Update()* returns an error if the specified node ID is not in the portal cache, or if the cached configuration is incompatible with the given command (e.g. attempting to remove a component that does not exist). Otherwise an asynchronous ‘request ID’ is returned which will be used later to report to the client (via *clientEndpoint*) the status of this update command.
— **void addListener(String wsEndpoint)** adds a ‘listener’ owned by a client that will be immediately notified of the occurrence of any change to the portal cache. Two types of change are reported: the appearance/disappearance of nodes to/from the WSAN; and updates of the software configurations of currently-known nodes. When a listener first connects, it is provided with the full content of the portal cache. Multiple listeners can be added to a portal; *void removeListener(String wsEndpoint)* is used to remove a previously-added listener.
— **void refresh(ID nodes[])** invalidates the portal cache entries of the specified nodes (or all nodes if desired), and asks SSP to re-acquire this information from the WSAN.

Clients use *update()* to initiate instructed evolutionary changes to the WSAN’s software, and *addListener()* to register their interest in being informed of per-node software configuration changes as they take place. The portal cache is asynchronously updated whenever a software configuration change is reported to the portal by SSP. The *refresh()* operation explicitly flushes the portal cache and causes it to be repopulated anew by SSP. Cache entries are tagged with a unique *version hash* that is generated by nodes when their configuration changes and sent to the portal via SSP. This version hash is used by the portal when sending software update commands via SEP; SEP commands are accepted by nodes only if the hash matches their current version hash. This mechanism ensures idempotence of updates as well as assuring that the portal is taking action on the basis of up-to-date information for a given node.

A key property of the portal API is that all of its operations are *asynchronous* – i.e. they return immediately without blocking. This acknowledges the fact that updates may take a very long time to complete (tens of minutes) due to the extreme resource constraints of WSANs. The asynchrony also makes it straightforward to *federate* portal API instances to facilitate managing several isolated WSANs as a single entity. This can easily be achieved by deploying a ‘federator’ portal that transparently makes multiple WSANs appear as one by multiplexing the listener reports of individual per-WSAN portals into its own cache and dispatching updates to the appropriate portal(s).

5.2. Clients

Clients make function calls on the portal API and also implement a client-side API through which the portal updates the client when changes occur in the WSAN. The client API is as follows:

— **int updateStatus(int reqID, int status)** reports on the outcome of a previously-issued *update()* command as associated with the given request ID. Using the *status* parameter the portal reports one of either SUCCESS, FAILURE or TIMEOUT. The TIMEOUT status is used when the portal cannot be certain whether an update succeeded or not (essentially this is analogous to the case in traditional networking whereby the sender cannot be sure whether a message was lost or is still 'in flight').
— **void nodesUpdated(ID nodes[], Configuration[] configs)** reports new software configurations of nodes and/or that the nodes exist. If the TIMEOUT case was reported for an *update()* command, the client (or end-user) can follow its own decision process regarding the success or failure of an update by observing the respective node's software configuration over a longer time period.
— **void nodesLost(ID nodes[])** reports that the given nodes are no longer considered to be part of the network.

Using the above APIs clients can gain a good level of certainty about software updates despite the underlying unreliable medium. Note that the primary TIMEOUT status is reported by the portal, instead of being independently presumed by the client, because the portal API's *update()* operation may not immediately dispatch the command into the WSAN but rather may enqueue it behind other update requests. TIMEOUT from the portal therefore indicates that the portal is no longer trying to deliver the update to the target node(s). FAILURE by comparison indicates that the portal never dispatched the command into the WSAN (perhaps because the node was considered failed at this point) or else has cause to be authoritative about the failure.

We generally envision clients being web-based and graphical in nature allowing users to interactively view the WSAN's current node population, inspect individual nodes' software, and drag and drop components onto nodes as desired.

6. PROTOCOLS

6.1. The overlay protocol (OLP)

The role of OLP is to provide the other three protocols (SSP, SEP and SDP) with lightweight, unreliable, bidirectional communication between the portal and the nodes in the WSAN. OLP offers this service only for messages that are small enough to fit into a single MAC frame. Like the rest of our protocol suite, the role that OLP plays is orthogonal to those of the other protocols and is therefore 'pluggable'; i.e. the role could be filled by various alternative implementations as long as they conform to the simple API defined below. While our lightweight OLP implementation does not in itself represent a significant contribution of our work we choose to briefly describe it here for the sake of completeness; the behaviour described in the following is that which we later use in the context of our evaluation. More advanced implementations of a similar service may be found in e.g. [Gnawali et al. 2009; Moeller et al. 2010]; our preference here is towards a simple reference implementation to in turn simplify our evaluation in terms of our remaining three protocols.

Our reference OLP implementation works essentially as a multi-channel tree-based overlay network that allows data to be sent from the portal down to nodes as well as from nodes up to the portal. Our implementation requires nodes only to maintain state relating to their current parent.

OLP's API on each node comprises the following calls:

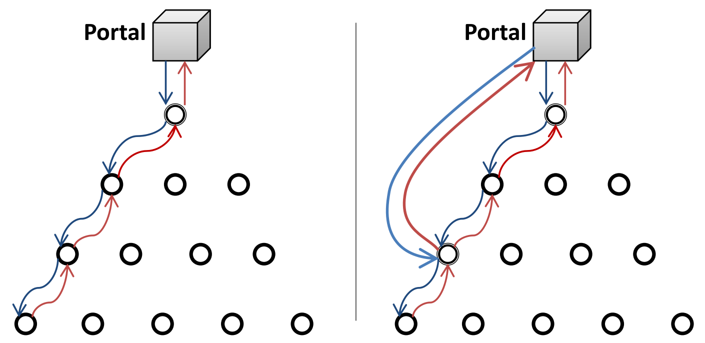


Fig. 3. The role of multiple gateway nodes in OLP. On the left is a single-gateway tree with messages travelling up and down the tree; the tree on the right features an additional Internet-connected gateway node to transparently improve performance in both upstream and downstream directions. As can be seen, the additional gateway node is reducing load on the upstream nodes and reducing the average distance between nodes and their portal.

— **void send(vc, data)** enables the 'user' (i.e. SSP, SEP or SDP) to send a message upstream towards the portal on a specified 'virtual channel' vc.
— **void register(vc, cb)** enables the user to register a callback cb with the signature: *message callback(vc)*.

The callback is used to receive messages on a specified vc from both upstream and downstream directions. In the upstream case, the message will have been issued using *send()* on the specified virtual channel and in the downstream case by piggy-backing data on an OLP 'tree maintenance packet' (see below).

OLP incurs low traffic overhead and holds minimal state. Nevertheless it is designed to work efficiently with ad-hoc network topologies of arbitrary scale. The protocol builds and maintains a simple tree-based network overlay: every $P_{olp}ms$ the portal sends a *tree maintenance packet* with a sequence number to the WSAN. Each time a node within the WSAN receives one of these packets that has a higher sequence number than the last one it received, it marks the sender of that packet as its 'parent' and forwards the packet using broadcast. This process continuously refreshes the tree structure and automatically accommodates churn. Note that nodes do not need to keep any information about their descendants – in accordance with our principle of minimising state information.

Downstream messaging over OLP is highly limited in capacity since tree maintenance packets have limited space for piggybacked data (100 bytes in our implementation); furthermore these messages are only sent every $P_{olp}ms$ and their delivery to any node is unreliable. The higher-level protocols (i.e. SSP, SEP, SDP) that use OLP are explicitly designed to operate with good performance within these limitations.

Finally, it is evident that a gateway-based architecture like that illustrated in Fig. 2 has inherent scalability limitations when supporting portal $\leftrightarrow$ node communication. In particular, the gateway is a bottleneck both for incoming SSP reports and for outgoing SDP data. To address this issue, OLP and our remaining protocols are designed to operate in the context of a WSAN architecture that can optionally incorporate multiple gateways that are strategically deployed throughout the network. This essentially enables us to scale arbitrarily by avoiding the single bottleneck incurred by a single-rooted tree and reducing the average distance between nodes and their portal.

Additional gateway nodes appear to OLP (and therefore also to SSP, SEP and SDP) as entirely normal nodes *except* that they have direct two-way Internet connectivity to the portal. They thus have the capability to relieve the load on the main gateway, without the associated state requirements of maintaining multiple trees, by optionally and transparently inserting special nodes into the OLP tree. Additional gateways of this nature can be added and removed dynamically to improve overall performance without impacting behaviour. This is illustrated in Figure 3.

Additional gateway nodes are transparent such that whenever a new tree maintenance packet is due to be sent out by the portal with a new sequence number, this packet is sent near-simultaneously from the main gateway and all additional gateway nodes using the same sequence number; nodes will thus assign as their parent the node from which they first receive this sequence number, ignoring any following receptions of this sequence number. All upward traffic received by an additional gateway is similarly forwarded direct to the portal rather than through the wireless network.

6.2. The software status protocol (SSP)

SSP is responsible for maintaining the portal's soft-state cache of the WSAN's current node population and software landscape. Whenever a node sees a change in its software configuration, whether initiated from the portal or autonomously by the node itself, the SSP protocol component on that node sends information on the new configuration towards the portal (i.e. to their parent) in a stream of *report packets*. This is done using the OLP component's *send()* API as discussed in Section 6.1. SSP components also forward any received report packets to their parent.

Viewing our overall approach as a sensor / actuator system for software itself, SSP is responsible for the 'sensing' aspect, gathering each node's current software configuration. The kind of sensor data involved is unusual however. Firstly, during stable periods it is not a continuous stream of values but rather a fixed value representing a node's current configuration. Secondly, this fixed value occupies several thousand bytes of non-compressible data for each sensor node and it must therefore be delivered in a fragmented fashion.

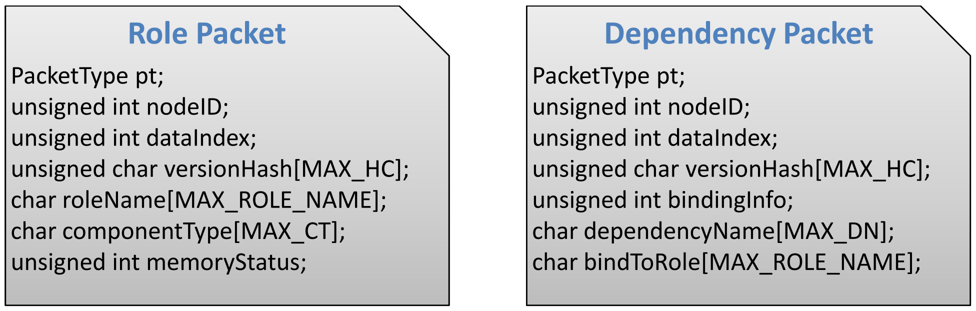
SSP's job is therefore, each time a node's software changes, to deliver this fixed value for each node to the portal in small fragments over an arbitrary number of unreliable hops, with any lost packets in a node's data series retransmitted as required.

6.2.1. The SSP packet format. SSP uses a stream of 48-byte packets to report a node's software configuration; where possible multiple such packets can be included in one underlying MAC frame. We use two such packet variants: *role packets* and *dependency packets* as shown in Fig. 4. Both variants have a standard upper part which contains a packet type field; the reporting node's ID; the version hash corresponding to the node's current software configuration; an integer representing the total number of packets that describe the node's configuration; and an integer representing the packet index into this total that the current packet represents.

Given this information it is possible for the portal to determine from any SSP packet whether the node's overall software configuration has changed because the portal will observe a new version hash. The version hash is also used by SEP when instructing a node to make changes to its software (see Sec. 6.3).

The lower parts of the two packet variants then differ as follows: The **role** variant is used to describe a component in the system and contains the string name of the component instance it is describing and a byte sequence that identifies the component type from which this instance is sourced. This byte sequence is a string that uniquely identifies the object file of the component type (for example a hash of the object file's contents). Finally role packets provide information on the node's available memory.

The **dependency** variant is used to describe a single dependency of one role on another role and whether or not this dependency is currently satisfied. Its bindingInfo field notes in the high-byte the data index of the role that owns this dependency while



Role Packet

```
PacketType pt;
unsigned int nodeID;
unsigned int dataIndex;
unsigned char versionHash[MAX_HC];
char roleName[MAX_ROLE_NAME];
char componentType[MAX_CT];
unsigned int memoryStatus;
```

Dependency Packet

```
PacketType pt;
unsigned int nodeID;
unsigned int dataIndex;
unsigned char versionHash[MAX_HC];
unsigned int bindingInfo;
char dependencyName[MAX_DN];
char bindToRole[MAX_ROLE_NAME];
```

Fig. 4. The two 48-byte packet formats used by SSP on the TelosB platform to describe roles and their dependencies. Any single packet serves either to announce the node's presence or to indicate that its software has changed with a new hash code. The high byte of *dataIndex* contains the total number of data items, and the low byte contains the index into that total that this packet represents.

the low byte indicates whether or not it is currently satisfied. The dependencyName field then contains a string representing the required interface type of the dependency; and the bindingName field contains the string role name of the component that this dependency is expected to be satisfied against. Note we cannot use an index for the latter information because the role may not actually exist in the system.

6.2.2. The SSP report cycle. The SSP implementation on each node continually executes a *report cycle* in which it sends a stream of report packets that collectively define the current software configuration of the node. The data is of necessity sent in this incremental manner because we are working directly over a MAC layer with small (110 byte) frames. Any single report packet represents discovery of the node itself.

In each report cycle the node iterates over the Lorien system manifest (i.e. over each of its installed components), sending report packets for each listed role to describe that role in itself and then its dependencies. The period P_{ssp} with which a node sends report packets is set to P_{ssp}^{min} whenever the node's software configuration changes, and set to P_{ssp}^{max} (typically a far longer period) each time a full description of the software configuration has been sent. At P_{ssp}^{max} just one report packet is sent per network packet in order to minimise energy expended; whereas at P_{ssp}^{min}, the protocol packs two successive report packets into one larger network packet to minimise latency at the portal.

Nodes running at P_{ssp}^{max} serve to provide occasional 'liveness pings' to the portal while avoiding undue congestion of the network.

6.2.3. Forwarding and buffering SSP packets. When an intermediate node receives a report packet it adds it to a buffer queue of size B_{ssp}^{len}, and every $B_{ssp}^{clr} ms$ it sends the first packet in this buffer to its OLP parent (which in turn will buffer the packet in its own queue). This classic store-and-forward mechanism caps each node's SSP workload to a maximum bytes-per-second rate and helps prevent packet storms in the network which might otherwise occur if nodes immediately forwarded all messages they received.

Because the buffer queue of a node is of limited size, we adopt a policy that has been found in practice to fairly distribute packet discards across nodes. The policy successively applies the following three steps whenever a new report packet is received and the queue is already full: i) if the queue already contains any other packet from the origin node of the incoming packet, drop the incoming packet; ii) otherwise, if the queue contains multiple packets from some other node, drop one of these packets to make room for the incoming packet; iii) otherwise drop a random packet selected from the union of the queue and the incoming packet.

6.2.4. Controlling SSP from the portal. As mentioned in Sec. 6.1, the portal can issue simple commands to nodes by piggy-backing information on OLP tree maintenance packets. SSP employs just one such command, REFRESH, which sets $P_{ssp} = P_{ssp}^{min}$ at the specified node(s), thereby providing faster information streams from those nodes. The portal can choose to refresh all nodes or a limited subset; and furthermore in the latter case can either request a complete refresh of a selected node or else a refresh of selected data indices in the packet sequence of that node. This latter case is useful to avoid having nodes resend an entire software configuration report when only a few SSP packets in the sequence are lost. It works by sending the data index D_i from which the selective refresh is desired along with an 8-bit pattern representing the indices that are needed from D_i onwards.

When the portal is first started up it issues a REFRESH command to all nodes to initiate the process of filling the portal cache. It then uses selective refresh to progressively fill in any knowledge gaps in the received data. The portal can also make assumptions about the liveness of a given node based on when a P_{ssp}^{max} heartbeat was last received; and if a new node appears that was previously not known to the portal cache then a selective node refresh is issued to that node. Note that the latency and reliability of REFRESH can be improved by issuing them via additional gateways (where available) as well as via the WSAN's main gateway node.

6.3. The software evolution protocol (SEP)

The purpose of SEP is to control software evolution on nodes. Viewing our overall approach as a sensor / actuator system for software itself, SEP is responsible for the 'actuation' aspect, making changes to each node's current software configuration. In achieving this we logically divide the component space of a node's software configuration into two regions: an 'instructed' region and an 'autonomous' region. This is done to harmonize these two forms of evolution. Note that autonomous decision making processes themselves are beyond the scope of this paper and left to future work.

6.3.1. Instructed Evolution. Software updates are effected at nodes by sending packets that encapsulate Lorien's software evolution commands (see Sec. 3), allowing the addition, removal and replacement of components and their interconnections. The key problem for instructed evolution – software updates specified directly by administrators or users – is guaranteeing that a desired software change happens exactly once.

As in the case of SSP, SEP commands are piggy-packed on OLP tree maintenance packets and delivered to targeted nodes. Each SEP command specifies the node to which the command applies, the most recent known version hash of that node (as indicated in the portal cache), a set of software evolution instructions (add/remove/replace), and a list of files on which those instruction depend. These files comprise code modules (i.e. executable components that must be present in external flash memory before the software evolution step can proceed) and Lorien configuration fragments.

When a SEP command arrives at a target node, the SEP component on that node first checks the command's version hash against the current value recorded at the node. If these differ, SEP drops the command on the grounds that the portal cache data on which it was based is stale. This behaviour ensures *idempotence* of updates.

Otherwise, SEP checks whether the required files are already present in the node's external flash memory. The expectation is that this will often be the case: this is true for example when adding a component that was previously added (and subsequently removed) at this node; and, depending on SDP's garbage collection policy, may also be true when a file has transited this node en-route to other nodes from earlier evolutions elsewhere in the network. If the required files are not present, SEP calls SDP to obtain them as will be discussed in Section 6.4. When SEP is subsequently informed by SDP that the modules have arrived, it enacts the required software updates using standard Lorien services. When the update is complete, the node's version hash is regenerated and SSP starts a new report cycle at P_{ssp}^{min}.

Note that where additional gateway nodes are available, SEP commands can be issued via these gateways as well as via the tree's root node. This will typically reduce latency and also improve reliability of delivery.

6.3.2. Autonomy Model. While we do not define in this paper the autonomy processes that select and learn how to optimally configure a node's software configuration, we do define a basic reference model for autonomy such that it may exist in synergy with the instructed evolution approach described above. Autonomy is defined here as the ability for nodes to independently add, remove and replace components and their interconnections as informed by locally available context at the node[3].

There are two key problems with autonomy: First, if autonomy has a completely free hand in terms of the node's software configuration, uncertainty about instructed evolution commands can arise when an instruction is successfully delivered by SEP and correctly enacted, but the change is immediately reversed by an autonomous agent with conflicting motives. This situation is indistinguishable from a failed instructed evolution, making the job of the portal server and clients more difficult.

Second, if we were to include in SSP reports components in the autonomous region we may see high degrees of churn as frequent autonomous changes are made. This would keep SSP running at P_{ssp}^{min}, thereby expending network resources; but more importantly it would mean frequent changes to a node's version hash. The latter pathology, if occurring frequently enough, could actually prevent instructed evolution commands from ever succeeding because they would always carry an old version hash and so be rejected as discussed above.

It is therefore convenient to have SSP and SEP operate exclusively over the 'instructed region' of a node's software configuration while a separate 'autonomous region' exists under the control of autonomy agents. This is illustrated in Fig. 5.

In more detail, in the component configuration initially flashed to a node (e.g. at the factory) all components are within the instructed region. Autonomy agents may add and remove components outside this region (such as new applications or network components) but may not make changes to components within the instructed region. This is illustrated on the left side of Fig. 5. When an instruction is issued via SEP to add a new component to a node this expands the instructed region; while an instruction via SEP to remove an existing component contracts the instructed region and correspondingly expands the autonomous region (shown on the right of Fig. 5).

The instructed and autonomous regions are strictly separated and the instructed region takes priority: hence if a component is added via SEP (i.e. to the instructed region), and it already existed in the autonomous region, the component is removed from the former and added to the latter. If however a role is removed by SEP the autonomy agent(s) are free to add that role back in but it will not show in SSP reports or be reflected in the version hash delivered by SSP.

Overall this reference model avoids SSP and version hash churn allowing SEP instructions to succeed despite high rates of autonomous change. It also guarantees that instructed evolution will not be 'undone' by autonomous evolution leading to ambiguity over whether an SEP instruction arrived and was executed at a node or not. Auton-

[3]This is 'compositional autonomy'; the other commonly identified dimension of autonomy, 'parametric autonomy', is less general and can be subsumed under compositional autonomy (see e.g. [Lorincz et al. 2008; Sengul et al. 2008] for approaches to parametric autonomy in WSANs).

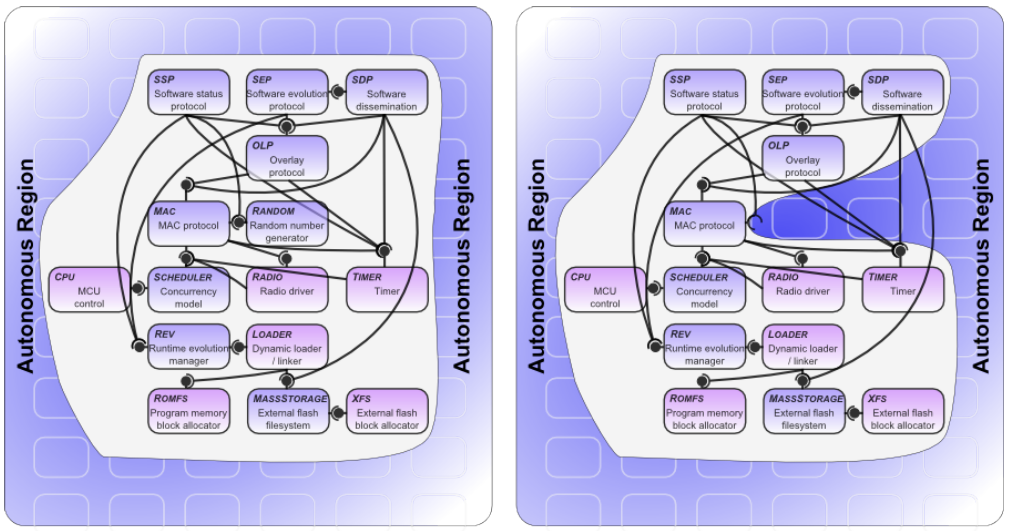

Autonomous Region
SSP Software status protocol
SEP Software evolution protocol
SDP Software dissemination
OLP Overlay protocol
MAC MAC protocol
RANDOM Random number generator
CPU MCU control
SCHEDULER Concurrency model
RADIO Radio driver
TIMER Timer
REV Runtime evolution manager
LOADER Dynamic loader / linker
ROMFS Program memory block allocator
MASSSTORAGE External flash filesystem
XFS External flash block allocator
Autonomous Region

Fig. 5. The instructed and autonomous regions of a sensor node's software composition. – The instructed region (as seen by SSP and controlled by SEP) is shown in white with the autonomous region surrounding it in blue shading. In the right side of this picture the Random role has been removed by SEP, expanding the autonomous region. This does not necessarily mean that there is no random number generator in the system; on the contrary autonomy managers will likely have as part of their rule set the condition that a role on which other roles in the system depend must be loaded.

omy can be disabled entirely on selected (or all) sensor nodes simply by using SEP to remove the autonomy manager component(s), or re-enabled by adding them.

6.4. The software dissemination protocol (SDP)

SDP is responsible for the dissemination of software modules and other files to nodes as required by the SEP implementation on those nodes. Dissemination of files must be reliable and error-free. SDP's API has one operation, *getFile()*, which is called by SEP if a needed file is not present.

Rather than design and implement an entirely new directed, and therefore divergence-friendly, file dissemination protocol our approach here was to instead examine the potential to extend an existing well-studied protocol with directed-dissemination capabilities. In this respect the most appropriate existing protocol of which we are aware is the well-known Deluge [Hui and Culler 2004] protocol which already works with the general paradigm of 'objects' to be disseminated. In this section we therefore discuss how the basic node-to-node file transfer protocol of Deluge is extended with a directed dissemination protocol.

Abstractly, our directed dissemination approach is designed as a kind of pull-based store-and-forward: a needed file is injected at the gateway node of the WSAN and then a node E is instructed via SEP to acquire this file. E asks its parent in the OLP tree for this file; its parent then asks its own parent in the OLP tree, and so on, until the gateway node is asked for this file. The gateway node then sends the file to its requesting child, which in turn sends the file to *its* requesting child, until the file reaches the original requesting node. Only nodes on the direct path between a requesting node and a source are therefore involved in its transfer.

To achieve this pull-based directed dissemination the SDP component on each node maintains three lists: WantedFiles, AdvertisedFiles and DownloadingFiles. In quiet conditions all three lists are empty such that SDP uses no network traffic. When SEP on a node invokes *getFile()* SDP adds the requested file to its WantedFiles list; each file on this list is requested from the node's parent in the OLP tree with a period of P_{sdp}^{wfa}; this period decays for each file by $P_{sdp}^{wfa\_decay}$ after each request up to a maximum of $P_{sdp}^{wfa\_max}$. This decay acknowledges that a needed file is increasingly more likely to be en-route as more requests for it are made by a given node and so repeated requests are increasingly redundant. We do not decay to more than $P_{sdp}^{wfa\_max}$ in case of heavy packet loss causing all previous requests to be lost. If a node receives a wanted file request from a child it checks if the file is present locally and if so adds that file to AdvertisedFiles; otherwise it adds the file to its own WantedFiles list.

Every P_{sdp}^{afa} SDP then broadcasts an advertisement for one file on its AdvertisedFiles list. Each file on this list is advertised C_{sdp}^{afc} times before being removed from AdvertisedFiles. Actual node to node file transfer then works very similarly to the Deluge protocol on which SDP is loosely based: files are divided into logical fixed-size pages, which are further subdivided into fixed-size packets. A node advertising a file that it holds locally advertises the 'highest' page of that file, meaning the page containing the data either at, or else the page which is most towards, the end of the file. If a node receiving an advertisement has this file on its WantedFiles list it adds that file to its DownloadingFiles list and initiates a Deluge-like file transfer protocol over the packets and pages of the desired file until it has successfully downloaded the complete file.

In detail, this transfer proceeds by a requesting node R asking an advertising node A for the lowest missing page of an advertised file that it wants locally, A then sending each packet of that page in sequence, and R requesting re-transmission of any packets in the sequence that it missed for that page. Both packets and pages have CRCs applied to them which are checked at R on reception of a packet or complete page.

SEP is informed by SDP when when each of its requested files have been fully downloaded, at which point SEP can enact its pending software evolution commands.

Note that where additional gateway nodes are available, popular software modules can be directly injected at all gateway nodes as well as the tree's root both to reduce the number of hops and to relieve the bottleneck at the gateway and near the root of the OLP tree. Finally, SDP nodes that are along a forwarding path can apply various garbage collection strategies for files that are not currently wanted locally; they can for instance be deleted after some period or can be deleted once secondary storage is near-capacity (as mentioned in Sec. 6.3.1 it is possible that a component which transits through a node may at some point in the future be needed at that node itself).

The above strategy for divergence-friendly dissemination of files is selected based on its simplicity and zero-overhead when no file requests exist in the network. An alternative possible design may appear to be to maintain an active routing infrastructure such as a second tree in which each node records not only its parent (as in OLP) but also a full list of its descendants. Given this, modules could be routed directly to target nodes. However, maintaining such a routing infrastructure would likely be costly in terms of protocol overhead even in quite small WSAN deployments, especially in the presence of significant node churn.

7. EVALUATION

In this section we evaluate our approach against the general goals set out in Sec. 1: *divergence* and *autonomy* are inherent properties of our approach described in the previous sections and not something that we empirically evaluate. The additional desirable properties described in Sec 1 are *minimality* of the approach, *unobtrusiveness* to sensor data and *energy efficiency* primarily in terms of network usage.

	OLP	SSP	SEP	SDP
Program memory	1420	2232	1170	4218
RAM	160	440	250	900

Fig. 6. Memory statistics measured in bytes for our four protocols on the TelosB platform (which has a total of 48KB of program memory and 10KB of RAM).

We interpret these three properties generally as the amount of *network data* consumed by our approach in various modes of operation such as receiving SSP reports or using SDP to disseminate new files to nodes to update their software. We also examine the general *performance* of our approach (e.g. time for a change to take effect).

In terms of comparative study there is little other work that we are aware of with which we can compare our results in a like-for-like manner (the majority of update work focuses on image-based updates as discussed in Sec. 2). We therefore approach our evaluation by using baseline values as comparison points where possible, taking a low-resolution sensor data stream as this baseline.

Our evaluation uses Lorien 2.8.4, compiled with mspgcc4.4.4, and uses Lorien's basic unreliable CSMA MAC protocol for all data communication. The software configuration running on each sensor node is exactly that shown in Fig. 1; in total **65** SSP packets are needed to fully describe a node running this software configuration, comprising 3120 bytes of information per node that need to reach the portal. All results reported here can be recreated from source at [Lorien Research Entry 2012]. Note that, although we use Lorien as our base OS throughout this evaluation, our results can be treated effectively as being OS-independent as we only measure communication specifically by the protocols described in this paper rather than any additional data that may/may not be sent by the underlying OS as part of its normal operations.

In the following sections we first describe our experimental setup and present a holistic view of the network characteristics and performance of our approach across different operating modes; we then examine in detail SSP and SDP to demonstrate their characteristics and performance as we scale up their workloads. We do not explicitly examine OLP and SEP in our evaluation: OLP is running throughout all experiments and thus contributes to the general networking overheads of our approach which are included as part of all of our measurements; meanwhile the use of SEP is essentially 'free' since SEP packets are piggy-backed on OLP packets.

Raw memory occupation is also however an important factor in low-power WSAN platforms; Fig. 6 therefore shows statistics for all four protocols, demonstrating that low overall memory usage is possible despite the advanced functionality provided.

7.1. Experimental Setup & Results Overview

7.1.1. Experimental Setup.

Our evaluation uses the Cooja WSN simulator [Osterlind et al. 2006] configured to emulate each node at the instruction level using the mspsim emulator for the TelosB platform. Using instruction-level emulation provides us with high fidelity results that are close to reality yet are still repeatable. Before embarking on our simulation-based evaluation, however, we examined the characteristics of our implementation on a small 15-node TelosB testbed and verified that observed behaviour is indeed very similar to what we observe in emulation for the kinds of experiments that we perform.

We configure the Cooja simulator using the UDGM radio model with a transmission range of 50m and an 'interference range' of 100m. Each topology we use is a square grid with a separation of 30m between nodes in topologies we designate as 'dense' (nodes having 8 neighbours on average), and 45m in topologies we designate as 'sparse' (nodes

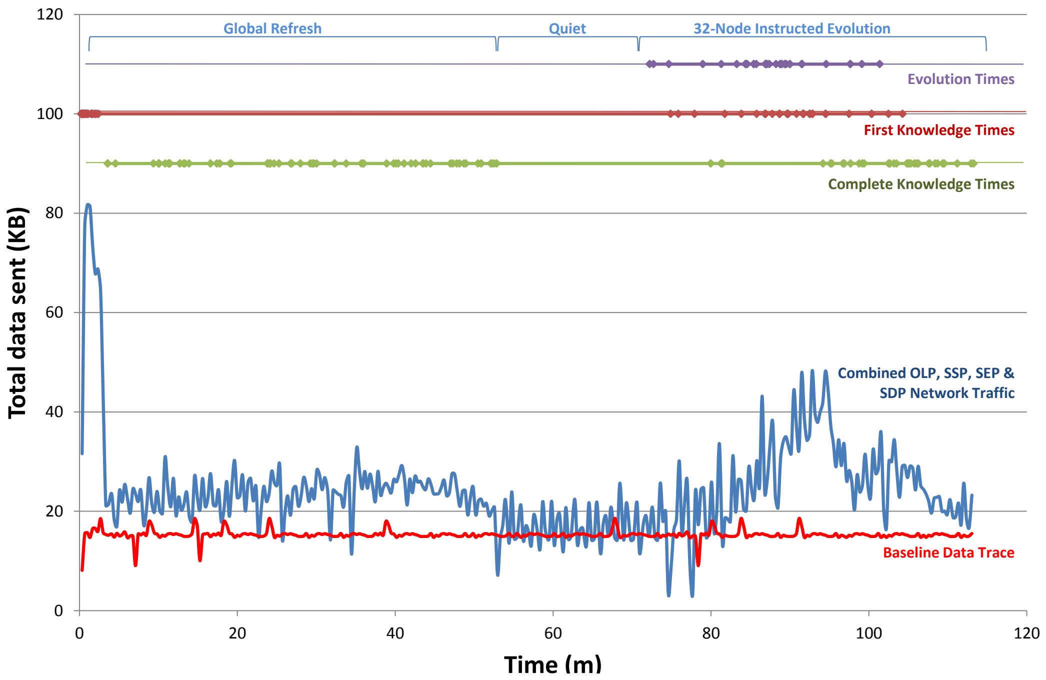


Fig. 7. Overview of network activity in a 64-node network in response to (i) a global refresh from the portal, (ii) a quiet period in which no software is evolving, and (iii) an instructed software evolution to add a new component to 32 nodes (including directed dissemination from the gateway node of this component type and a Lorien configuration fragment). Data plots are aggregated across the entire network with each point representing the last 20 seconds of total traffic; a point value of 20,000 bytes can therefore be interpreted as 1,000 bytes sent per second (on average) by all nodes in the network, or 15 bytes per node per second.

having 4 neighbours on average). Throughout our evaluation we therefore use network sizes that are workable as square grids: 9 nodes, 16 nodes, 25 nodes, etc., up to a maximum of 100 nodes which we found to be the effective limit of the simulator when operating in emulated mode on our 2Ghz Dual-Core machine with 2GB RAM. Each topology has a single gateway node and all experiments were repeated twice such that in one run the gateway is placed at the corner of the network and in the other run the gateway is placed at the centre. The results that we present are thus averaged across these four topology variants (dense/sparse, corner/centre) for each network size.

The way in which a portal uses our protocols is configurable and, throughout all of our experiments, was configured as follows: with respect to SSP, for each node that the portal knows about, but for which it currently has incomplete software configuration information, it waits for a period of $P_{server}^{ssp\_timeout}$ after each received SSP packet; if no further SSP packet is received within that time it issues a selective REFRESH command to that node indicating the bit-pattern that is missing (see Sec. 6.2). After issuing a REFRESH, $P_{server}^{ssp\_timeout}$ is reset for that node. $P_{server}^{ssp\_timeout}$ was configured at 30 seconds for all experiments. With respect to OLP, the periodic flooding of tree maintenance packets from the portal (i.e. P_{olp}) was set to a period of $10,000ms$.

7.1.2. Results.

Fig. 7 shows a detailed holistic view of the network characteristics and performance of our approach under different conditions in a 64 node dense emulated network with the gateway node at the corner. The system starts with a global REFRESH period (labelled ‘Global Refresh’) to gather the current node population and software landscape, followed by a quiet period ('Quiet') in which no evolution is taking place, and then followed by an instructed evolution period ('32-Node Instructed Evolution') in which a new application-level component is installed on 32 nodes. This installation involves the dissemination of both the component type object file (of size 700 bytes for this experiment) and the configuration fragment describing an instance of this component type to be added to the running systems of each of these 32 nodes.

The graph also shows a baseline data volume to provide a reference point. This is derived from a simple collection tree protocol running in the network in which each node sends its temperature value to the sink every 5 seconds (with no aggregation or other stream processing occurring). This collection tree uses a similar store-and-forward buffering strategy to SSP as described in Sec. 6.2, with random packet drops when a node's buffer is full. This represents a low-resolution sensor trace in which each sensor reading packet sent towards the portal is just 4 bytes; it serves to demonstrate that our approach performs reasonably well against a minimal baseline.

Looking at the results, during the quiet period the overall data rate drops to a level very close to our baseline (the data rate at this point is composed purely of SSP and OLP data since SDP is silent when no files are being disseminated and SEP is silent when no evolution commands are being sent) and rises to just 2 times this level at the peak of file dissemination via SDP in the instructed evolution phase.

The graph provides a good indication of overall performance. 'First knowledge times' relate to SSP's discovery of a node; this occurs relatively quickly with all nodes known about within 2.4 minutes of the global refresh, at an average of 80 seconds per node. 'Complete knowledge times' relate to SSP's delivery of information on the full software configuration of a node; this takes longer, with the portal attaining complete knowledge of all nodes in the network after 53 minutes, at an average of 26.5 minutes per node.

Finally, 'evolution times' provide an indication of SDP's performance; they demonstrate that the two files being disseminated are delivered to all 32 interested nodes within 33 minutes, at 16.5 minutes on average. Following software evolutions, the 'first knowledge times' and 'complete knowledge times' occur in parallel with continued file dissemination as SSP at the evolving nodes is progressively triggered into rapid status delivery by the change of node software.

7.2. Software Status Protocol

7.2.1. Setup and Metrics.

We now turn to a detailed examination of SSP in isolation to demonstrate its characteristics as we scale up its workload in networks of increasing size.

Our evaluation metrics for SSP comprise one network-related component (representing its energy consumption) and two performance-related components:

(1) **Data rate**: the average data rate (bytes/s) seen by a node during the course of establishing complete node population and software knowledge of a WSN.
(2) **Time to knowledge of a node**: the average time that it takes for the portal to become aware of a node in the network, as measured from the time a network-wide REFRESH command is issued.
(3) **Time to complete configuration knowledge**: the average time that it takes, under the same condition, for the portal to cache complete knowledge of the software configurations of nodes in the network.

The data rate metric here is a composite including OLP maintenance packets, SSP report packets, and SSP selective REFRESH packets issued by the portal via OLP when $P_{server}^{ssp\_timeout}$ timeouts occur as described in Sec. 7.1. In our experiments, we configured SSP's parameters to values that were found empirically to work well: $P_{ssp}^{min} = 5s$, $P_{ssp}^{max} = 60s$, $B_{ssp}^{len} = 4$ and $B_{ssp}^{clr} = 512ms$ (see Sec. 6.2 for details of these parameters).

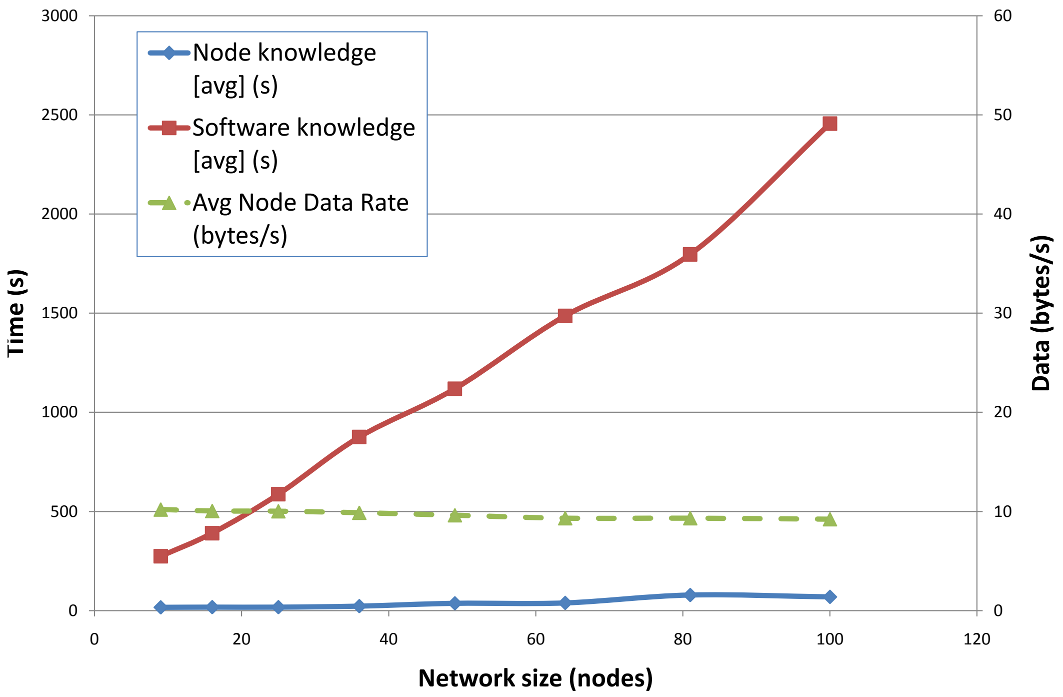


Fig. 8. Results from SSP experiments starting from a 'global refresh' command using varying network sizes. Data is this graph is the result of averaging data from each of the four topology variants described in Sec. 7.1 for each network size.

7.2.2. Results.

Fig. 8 shows results for the above metrics for increasing scales of network, from 9 nodes to 100. For each size of network the points on the graph are averaged over the four topology variants (dense/sparse, corner/centre) described in Sec. 7.1.

Taking the three metrics in turn, metric **(i)** (the average data throughput of each node) in all experiments is just 9 bytes per second including both OLP and SSP data. This remains generally fixed as we scale up thanks to the SSP buffering strategy described in Sec. 6.2 which enforces a maximum constant data rate at a given sensor node. We note that this, as will be discussed in Sec. 7.4, is not a *mandatory* data rate that must be upheld but is simply the rate required to achieve the performance reported here in terms of node and software knowledge times; SSP is by design a best-effort protocol with no inherent data rate or timing constraints.

Turning now to performance characteristics, metric **(ii)** (average time to knowledge of a node) is low throughout all tests, peaking at a maximum of just 118 seconds (2 minutes) on average to discover a node in the worst case of a dense 100 node network with the gateway node positioned at the corner.

Finally, examining metric **(iii)**, the average time to gather complete information about a node's software configuration is larger but scales with a linear trend peaking at 2500 seconds (42 minutes) in the largest of our experiment networks.

These results demonstrate that SSP does indeed operate with the characteristics of a minimally-interfering background traffic protocol irrespective of network size while maintaining performance characteristics that scale well with increasing network size.

To put this another way we can observe that SSP's relationship to our baseline in Fig. 7 is constant while performance remains good at scale.

7.3. Software Dissemination Protocol

7.3.1. Setup and Metrics.

In this section we examine our SDP implementation in detail to evaluate its characteristics as we scale up its workload in two different dimensions: firstly by disseminating increasing sizes of file to a fixed number of nodes in a fixed-size network; and secondly by disseminating a fixed size file to increasing numbers of nodes within a fixed-size network. In these more complex and time-consuming experiments we use emulated network sizes of 64 nodes. We also remind the reader that SDP is completely silent when no files are being requested of it (see Sec. 6.4), adding nothing to ambient network traffic. Throughout all of the following experiments, in addition to OLP, SSP was also running on each node at P_{ssp}^{max} as would be the case in a real deployment.

Before we introduce our metrics and results, it is important to understand the general characteristics of SDP in a modular operating environment. As described in Sec. 6.4, files needed by SDP are injected into the gateway node by the portal and then pulled towards the sensor nodes that require those files. The SDP nodes in the sensor network that initiate this 'pull' are those that are in turn called on by SEP to acquire files needed by an SEP evolution command issued to a given node. SEP's requirements usually involve either two files (a configuration fragment and a component type object file) in the case of adding a new component to a node's running software, or no files in the case of removing an existing component from a node's running software. The cost of the latter case is therefore effectively zero since the evolution command itself is piggybacked on an OLP maintenance packet that would have been sent anyway. The cost of the former case is *also* effectively zero if the necessary files are already present in a node's external flash chip (for example if those files have transited this node en-route to another downstream node, or else this component was once installed at this node at an earlier time). In this section we examine the plain performance of SDP without considering these special cases of virtually 'free' software evolution commands afforded by the modular design of the Lorien OS.

Under these conditions, our evaluation metrics for SDP comprise one network-related component (representing its energy consumption) and one performance-related component:

(1) **Data sent**: the total amount of SDP data needed throughout the network to disseminate a file to the selected nodes, including both data packets related directly to node-to-node file-transfer and data packets related to the directed dissemination of those files (see Sec.6.4).
(2) **Arrival time**: the average time that it takes for a file to arrive at a node that has requested it.

For all experiments, again based on values that were found empirically to work well, we configure SDP's parameters to be $P_{sdp}^{wfa} = 2000ms$, $P_{sdp}^{wfa\_decay} = 4000ms$, $P_{sdp}^{wfa\_max} = 60s$, $P_{sdp}^{afa} = 2000ms$ and $C_{sdp}^{afc} = 3$ (see Sec. 6.4 for details of these parameters).

7.3.2. Results.

Fig. 9 shows SDP's behaviour in the first domain of scalability where we are sending files of increasing size to a fixed number of nodes. The sizes of files vary from 700bytes, representing the size of a simple application-level component, to 5KB, representing a large component such as a file system. As mentioned above our network size is 64 nodes and 32 of those nodes are selected to download the file. This involves SDP at

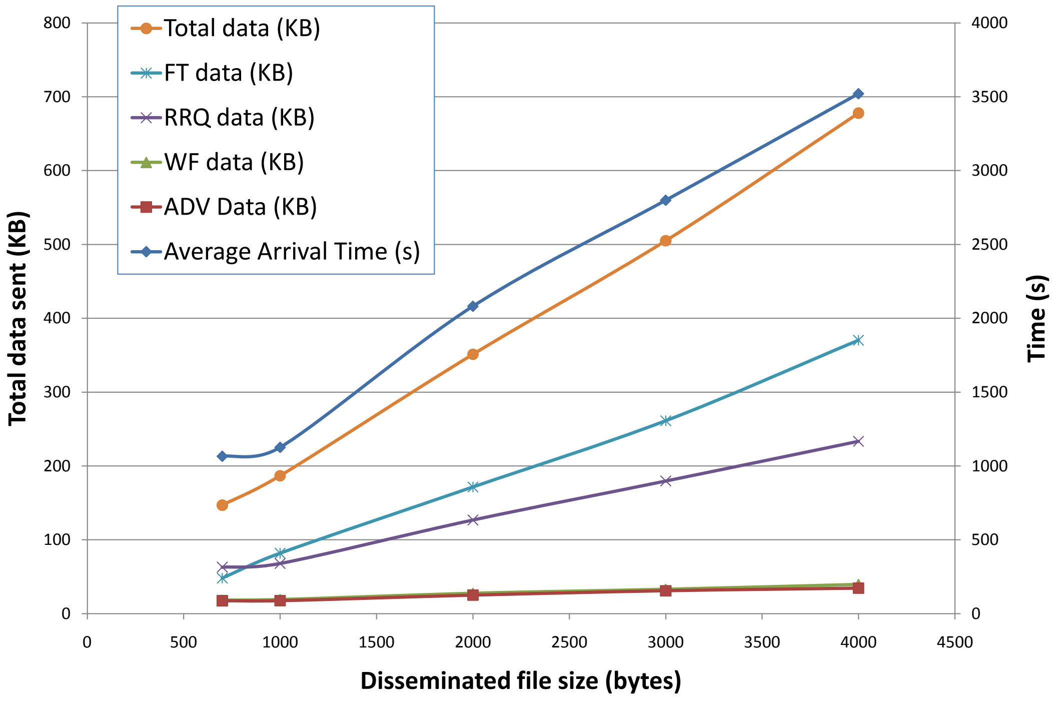


Fig. 9. Results from SDP experiments to disseminate files of increasing size to 31 nodes in a 64-node network. The network size figure includes the gateway node. – The graphs show the overall amount of data, and the composite parts of this data, needed by SDP over the entire network to disseminate these files to the selected nodes. SDP costs scale linearly with file size and across all experiments each sensor node on average sends data totalling two times the size of the file being disseminated.

each selected node sending 'wanted file' packets up the OLP tree and then involves nodes downloading 'advertised files' that they want or their siblings want. Again our results here are averaged across repeated experiments in the four topology variants described in Sec. 7.1.

In terms of metric **(i)** we see that the total data sent by SDP to deliver a file to all 32 target nodes increases linearly with increasing file size which is an encouraging trend. Fig. 9 also shows the component parts of this data, showing that file transfer ('FT') packets are the dominant component with re-request ('RRQ') packets being the second biggest contributor to overall data expenditure. By comparison wanted-file ('WF') packets and file advertisement ('ADV') packets, both used in directed dissemination, contribute minimally to overall data expenditure.

In terms of metric **(ii)**, Fig. 9 also shows the average time taken for a requested file to arrive at a requesting node, plotted on the secondary x-axis. This also has a linear trend as we increase the size of the file being disseminated.

Overall these results are positive although in future we intend to further investigate the potential to reduce the number of 'RRQ' requests for retransmission that are caused by missed file transfer packets.

Next, Fig. 10 shows SDP's behaviour in the second domain of scalability where we are sending a file fixed size to increasing numbers of nodes (i.e. divergent evolution). The size of file here is fixed at 3KB, representing an average component size, and our total network size is again 64 nodes, but in these experiments the number of nodes

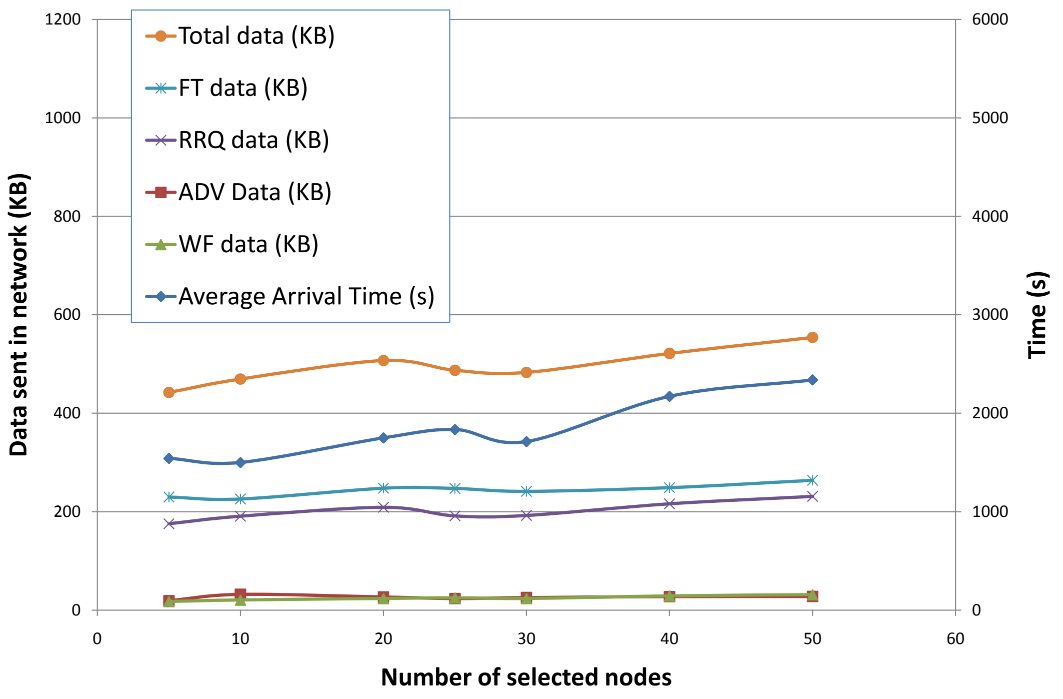


Fig. 10. Results from SDP experiments to disseminate a 3KB file to increasing numbers of nodes in a 64-node network. – Each point in the graph is an average over 4 runs with different randomly-selected nodes chosen to acquire the file. The graph shows the overall amount of data, the composite parts of this data, and the average time for a node to receive the desired file.

within that network that are selected to receive the disseminated file range from 5 up to 50. Each point on this graph is the average of four different executions using a different random seed to randomly select the nodes to which a file is disseminated from among the network's 64 nodes.

Examining metric **(i)** in this case we can see that the total data sent by SDP increases very slowly as more nodes need the file. From the component parts of this data we see again that FT and RRQ data are the main elements of data sent, and that deviations in RRQ traffic cause the minor deviations on the line of total data sent.

Examining metric **(ii)** in these experiments we see that the average time taken for a wanted file to arrive at a requesting node also increases slowly as we scale up the number of nodes wanting a file.

These results indicate that with just 5 randomly selected nodes the file tends to be disseminated to enough other nodes in the network (along the paths to those 5 nodes) that adding further nodes to the set wanting the same file is able to utilise those same paths to a large extent. Again this is a positive result indicating that SDP makes good use of the WSN's limited network capacity even without an active routing protocol underpinning it. We note that we have not analysed in detail the node-to-node file transfer involved in SDP's operation as this aspect of SDP is essentially based upon the existing well-studied Deluge protocol as discussed in Sec. 6.4.

7.4. Summary and Discussion

We have demonstrated in this section the characteristics of our approach and its ability to expend low resources comparative to a very modest baseline sensor data stream, with all parts of our approach resulting in an average data rate of just 9 bytes per sensor node per second during an SSP refresh. If desired this can be reduced even further by increasing P_{ssp}^{max} above the 60 second period used here, at the expense of potentially increased delay in noticing changes to a node's software configuration. We have also shown in detail the scalability of our SSP and SDP designs and the ability of SDP to efficiently support divergent software evolution in which sub-sets of nodes are selected for a new software configuration.

Finally, we reiterate that all parts of our approach are best-effort by design and are inherently able to accommodate reduced performance while still achieving their objectives. In particular SSP report packets are re-requested by the portal if missing; SDP disseminates files gradually and reliably; and our client and portal API design explicitly take into account the sometimes very slow response times of the underlying protocols due to their need to operate in low-power environments and maintain low baseline network traffic characteristics throughout.

8. CONCLUSION AND FUTURE DIRECTIONS

We have presented a comprehensive approach to managed software evolution for large-scale 'infrastructural' WSAN deployments which need to incrementally change/evolve their deployed software base on a routine basis. Our approach builds on safe component-based software updating with reflective capabilities as supported by the Lorien OS. The approach that we have presented here in particular features support for *divergent evolution* whereby the software configurations of different nodes can vary and evolve independently, and for *autonomy* whereby change can be initiated by nodes themselves as well as by centralised (instructed) command. The approach is based on a suite of four lightweight protocols that conspire to manage the software evolution process with low overhead and a minimal infrastructure requirement. Outside the WSAN, our approach involves a portal that exports a generic web services API to pluggable client programs. Thanks to our soft state approach, neither the portal nor the client need to maintain configuration files and can be freely attached and detached from the API at the user's convenience. We see this as an essential step towards a future of plug-and-play sensor networking.

Our evaluation shows that our approach exhibits low network overhead in comparison to a modest baseline and that SSP and SDP scale well as we increase both the size of the network and the size of the files being disseminated.

In the future, we intend to deploy our approach in the context of a city-scale WSAN infrastructure (specifically in the SmartSantander project [Bernat 2010]) to evaluate it in a real 'infrastructural' WSAN environment featuring distinct co-existing applications, multiple simultaneous users and the consequent need for frequent changes to the WSAN with minimal overhead and disruption. Finally, we intend to explore the autonomy domain in terms of sensor nodes making decisions about which components they should employ in which roles as their context and requirements change. We also plan to develop an online searchable library of components that can be downloaded in their compiled form for different hardware platforms.

Acknowledgements

This work has been partially supported by the European Union under contract number FP7-257992 (Smart Santander).

REFERENCES

BERNAT, J. 2010. SmartSantander: The path towards the smart city vision. In *Proceedings of the 1st ETSI M2M Workshop*.

BROUWERS, N., LANGENDOEN, K., AND CORKE, P. 2009. Darjeeling, a feature-rich vm for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. SenSys '09. ACM, New York, NY, USA, 169–182.

BUETTNER, M., YEE, G. V., ANDERSON, E., AND HAN, R. 2006. X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems*. SenSys '06. ACM, New York, NY, USA, 307–320.

CAO, Q., ABDELZAHER, T., STANKOVIC, J., AND HE, T. 2008. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*. IEEE Computer Society, 233–244.

CAO, Q. AND STANKOVIC, J. A. 2008. An in-field-maintenance framework for wireless sensor networks. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*. Springer-Verlag, Berlin, Heidelberg, 457–468.

DUNKELS, A., FINNE, N., ERIKSSON, J., AND VOIGT, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*. Boulder, Colorado, USA, 15–28.

DUNKELS, A., GRONVALL, B., AND VOIGT, T. 2004. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*. IEEE Computer Society, 455–462.

GNAWALI, O., FONSECA, R., JAMIESON, K., MOSS, D., AND LEVIS, P. 2009. Collection tree protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*. SenSys '09. ACM, New York, NY, USA, 1–14.

HAN, C.-C., KUMAR, R., SHEA, R., KOHLER, E., AND SRIVASTAVA, M. 2005. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. Seattle, Washington, USA, 163–176.

HILL, J., SZEWCZYK, R., WOO, A., HOLLAR, S., CULLER, D., AND PISTER, K. 2000. System architecture directions for networked sensors. *SIGOPS Operating Systems Review 34,* 5, 93–104.

HUGHES, D., GREENWOOD, P., BLAIR, G., COULSON, G., GRACE, P., PAPPENBERGER, F., SMITH, P., AND BEVEN, K. 2008. An experiment with reflective middleware to support grid-based flood monitoring. *Concurrency and Computation: Practice and Experience 20,* 11, 1303–1316.

HUI, J. W. AND CULLER, D. 2004. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. ACM, 81–94.

HUYGENS, C., HUGHES, D., LAGAISSE, B., AND JOOSEN, W. 2010. Streamlining development for networked embedded systems using multiple paradigms. *IEEE Softw. 27,* 5, 45–52.

LEVIS, P. AND CULLER, D. 2002. Maté: a tiny virtual machine for sensor networks. *SIGOPS Operating Systems Review 36,* 5, 85–95.

Lorien Research Entry 2012. http://lorien.xrna.net/research/?id=tosn2012porter.

LORINCZ, K., CHEN, B.-R., WATERMAN, J., WERNER-ALLEN, G., AND WELSH, M. 2008. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. ACM, New York, NY, USA, 211–224.

MOELLER, S., SRIDHARAN, A., KRISHNAMACHARI, B., AND GNAWALI, O. 2010. Routing without routes: the backpressure collection protocol. In *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. IPSN '10. ACM, New York, NY, USA, 279–290.

OSTERLIND, F., DUNKELS, A., ERIKSSON, J., FINNE, N., AND VOIGT, T. 2006. Cross-level sensor network simulation with cooja. In *Local Computer Networks, Proceedings 2006 31st IEEE Conference on*. 641 –648.

PANTA, R. K., BAGCHI, S., AND MIDKIFF, S. P. 2011. Efficient incremental code update for sensor networks. *ACM Trans. Sen. Netw. 7*, 30:1–30:32.

PISSIAS, P. AND COULSON, G. 2008. Framework for quiescence management in support of reconfigurable multi-threaded component-based systems. *Software, IET*, 348–361.

PORTER, B. AND COULSON, G. 2009. Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. MidSens '09. ACM, New York, NY, USA, 7–12.

PORTER, B., ROEDIG, U., AND COULSON, G. 2011. Type-safe updating for modular WSN software. IEEE Distributed Computing in Sensor Systems (DCOSS).

REIJERS, N. AND LANGENDOEN, K. 2003. Efficient code distribution in wireless sensor networks. In *2nd ACM International Conference on Wireless sensor networks and applications*. 60–67.

SENGUL, C., MILLER, M. J., AND GUPTA, I. 2008. Adaptive probability-based broadcast forwarding in energy-saving sensor networks. *ACM Trans. Sen. Netw. 4*, 6:1–6:32.

SMITH, P., HUGHES, D., BEVEN, K., CROSS, P., TYCH, W., COULSON, G., AND BLAIR, G. 2009. Towards the provision of site specific flood warnings using wireless sensor networks. *Meteorological Applications 16,* 1, 57–64.

STEFFAN, J., FIEGE, L., CILIA, M., AND BUCHMANN, A. 2005. Towards multi-purpose wireless sensor networks. In *ICW '05: Proceedings of the 2005 Systems Communications*. IEEE Computer Society, Washington, DC, USA, 336–341.

VARIOUS. 2011. International ic11 workshop on the intelligent campus. www.intelligentcampus.org/IC11/.