Application Interaction Model for Opportunistic Networking

Ramon S. Schwartz, Hylke W. van Dijk and Hans Scholten
Pervasive Systems Group, University of Twente, The Netherlands
{r.desouzaschwartz,h.w.vandijk,j.scholten}@utwente.nl *

Abstract

In Opportunistic Networks, autonomous nodes discover, assess and potentially seize opportunities for communication and distributed processing whenever these emerge. In this paper, we consider prerequisites for a successful implementation of such a way of processing in networks that consist mainly of heterogeneous devices. Devices are heterogeneous in size, in abilities, in movement, and in the role they play in the application.

The focus here is on the interaction at different levels and among various nodes, in view of our current scenario, where mobile nodes connect clusters of wireless sensors. The combined networks form an infrastructure-less sensor and actuation network. We propose a RESTful interaction model, which we demonstrate with an example implementation.

Keywords: *Opportunistic Networking, Interaction Model, Infrastructure-less Wireless Sensor Network, REST, Delay Tolerant Networks.*

1 Introduction

Sensor and actuator networks (SAN) may play a significant role in environment-monitoring applications. Typical examples include: air-quality and air-pollution monitoring, weather monitoring, embankment monitoring, and forest or hay fire monitoring. One specific application is that of an *early warning* system, which detects hazards or anomalies in an early stage.

The set of wireless, infrastructure-less networks, is a particularly interesting subset of SANs, since they potentially support cheap and lean implementations. Such networks can be flexibly deployed, and still achieve accurate and high quality monitoring. The fact that infrastructureless networks can be deployed on-demand in location and time is a great asset.

In this paper, we address the design of infrastructureless SANs by considering heterogeneous devices, which vary in size, in abilities, in movement, and in the role they play in the network. We consider nodes which act autonomously and opportunistically. Nodes interact with their environment, seizing opportunities for communication and distributed processing whenever they appear. Obviously, since node resources are limited, they must carefully select the opportunities they pursue. These interactions are referred to as Opportunistic Networking (OppNet).

Opportunistically operating nodes execute a Discover–Assess–Seize cycle. First of all, a node must be able to *discover* emerging opportunities for communication, (remote) processing, and (remote) storage. Second, nodes must be able to *assess* the costs and benefits of taking an opportunity and the chance the opportunity will prevail. Finally, a node will *seize* and configure the opportunity. Typically, a node must (re)configure its environment to suit its objectives, and thus increase the odds of a success for the taken opportunity. For each step in this cycle, different strategies exist. Discovering opportunities can range from gossipping [11] to a statically scheduled configuration [6]. Likewise, one can simply take any opportunity to disperse information or wisely assess the interests [2] of users in order to save node resources. Finally, seizing an opportunity implies (re)configuring the components involved in the communication, processing and storage of the associated action. Reconfigurations may be ad-hoc and unmoderated, like it is found in traditional peer-to-peer networks or reputation-based.

The Discover and Seize phases of the opportunistic networking cycle require a close interaction among nodes, processes and services. Interaction is required at each of these levels in an environment where nodes differ in capability and resource availability. In view of the application area of early warning systems for environment monitoring, it is imperative that nodes operate in an energy efficient way and act timely in case of anomalies.

In this paper, we explore a range of existing interaction models and assess their suitability for use in the context of infrastructure-less opportunistic networks. Our method classifies different levels of interactions. Given the application requirements, and the detailed requirements that stem from lower level interaction models, we opt for a in-depth comparison of two interaction concepts at the application level: WS-* and REST. We argue that scalability is the distinguishing requirement. Subsequently, we demonstrate the suitability of the selected interaction

*This work is supported by iLAND Project, ARTEMIS Joint Undertaking Call for proposals ARTEMIS-2008-1, Project contract no. 100026.

model (REST) for a concrete application.

The paper is organized as follows. In Section 2, we derive the principal requirements based on a general opportunistic network scenario. In Section 3, we classify existing interaction models and derive further requirements and suitable candidates in view of opportunistic networks. Following, we compare the sets of concepts and technologies of two service-oriented interaction model candidates in Section 4. Section 5 demonstrates the viability of REST. Finally, Section 6 concludes this work.

2 Requirements for OppNets

In our current work, we concentrate on the infrastructure-less SANS as illustrated in Figure 1, although the results are applicable for more general situations. We consider a range of connected technology islands. The heart of the network is an opportunistic network with moving nodes that provide a multi-hop delay-tolerant network with abundant resources. The predictability of nodes' mobility may range from a stable schedule pattern (bus) to a more unstable movement pattern in which the schedule of nodes exist but has to be relearned frequently (people, cars, bikes, etc.). At given locations, this opportunistic network interacts with technology islands through gateways. The gateways act as a centralized interaction points between the opportunistic network and the technology islands. Islands include a wireless sensor network (802.15), a wireless LAN (802.11), and an Internet backbone (802.3). In turn, these islands may deploy an opportunistic networking paradigm. Not shown in this figure is the ability of the opportunistic networking vehicle to use global wireless communications systems such as GPRS, UMTS, and satellite.

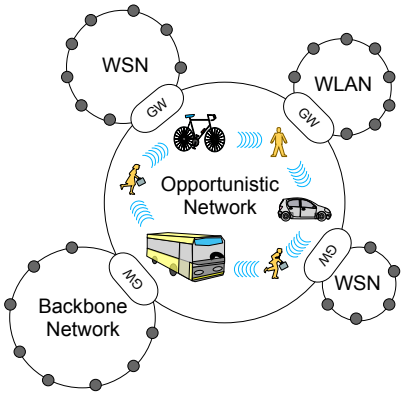

Figure 1: OppNet Scenario

Therefore, we focus on the interaction of applications or services in a widely heterogeneous environment with services residing in both powerful and limited (mobile) devices. The heterogeneity is also present in terms of predictability in mobility, and the role they play in the application. This scenario leads us to the following requirements:

- ***Scalability***: distributed applications must *scale* properly with an increasing number of nodes and interactions as well as with a high heterogeneity of device capabilities, since we envision a high number of nodes in the network environment, such as sensors, mobile phones, and servers.
- ***Resource Efficiency***: in every step of the Discover-Assess-Seize cycle, the utilization of resources must be efficiently, i.e., distributed among nodes in such a way that it does not compromise the overall functioning of the system. In each step in the Discover-Assess-Seize cycle, nodes will spend resources available both in the system and in the network. Upon a meeting, nodes have to exchange data and occupy the radio channel with certain bandwidth as well as spend energy in case of battery-powered devices. Storage is also of significant concern, since most data is constantly stored, carried, and forwarded.
- ***Reconfigurability***: the system experiences frequent node (re)configurations due to the constant reevaluation of opportunities, which aim to improve data delivery, distributed processing, and other goals the system might have. The presence of various underlying technologies employed by different applications and services brings the requirement of a platform-independent architecture with support for highly flexible reconfiguration.
- ***Security/Privacy***: OppNets inherit privacy and security problems from pervasive computing, as theoretically any communicating device could participate in the distributed application. Therefore, mechanisms to assure secure communication and privacy for the users of these devices are essential. In this work, however, we leave this requirement as an open issue.

3 Overview of Interaction Models

In this section, we classify interaction models into different levels, as shown in Figure 2. In the lower, processing level, the interaction between processes addresses the operations executed by each interacting entity. This interaction can be either synchronous or asynchronous. On top of that, the communication between partners can be also accomplished by means of synchronous or asynchronous interaction. One direct mapping of the synchronous communication model is the Remote Procedure Call (RPC), whereas Message-Oriented Middleware (MOM) is commonly associated with asynchronous communication. Client-server and Peer-to-peer are generally the main models for the network architecture. Finally, applications rely on design patterns such as publish/subscribe and/or Service-Oriented Architecture (SOA) to perform their interaction at application level.

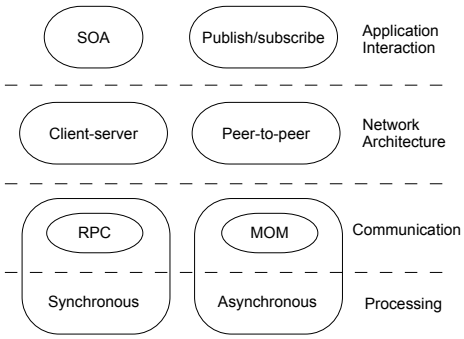

Figure 2: Interaction Model Classification

Synchronous

In a synchronous *processing* paradigm, the caller process blocks and waits (suspend processing) until the receiver completes its processing and returns control to it. This processing comprises all the operations performed by the operational system and network. From a *communication* perspective, a synchronous interaction implies that both communicating partners must be present at the moment the connection between them starts. That means that it is time coupled. We assume this distinction between *processing* and *communication* levels in the remainder of this work. Because of its nature, synchronous interactions benefit from the fact that lower and upper bounds for the process execution time can be set, considering unpreemptive scheduling. This is mainly due to the fact that (i) the data transmitted is received within a known bounded time; and (ii) drift rates between local clocks have a known bound. Consequently, synchronous distributes systems have the notion of global time, instrument predictable timing, and have a simple failure detection mechanism using timeout, which makes them suitable for hard real-time applications.

Asynchronous

In asynchronous *processing* paradigm, the caller process continues processing while the receiver completes the caller's request before it returns control. Consequently communicating partners do not need to be simultaneously present to start and interaction. Hence time is decoupled and the caller may continue processing regardless of the processing state of the invoked method. The actual exchange of requests is typically implemented by means of intermediary messages queues. Unlike synchronous interactions, in asynchronous interactions boundaries on the process execution time, data transmission delays or on drift rates between local clocks are often not guaranteed. Consequently, asynchronous distributed systems lack the notion of global time, there is only a logical ordering of events. Further, absolute timing is impossible, which renders the simple use of timeouts for failure detection tedious.

Remote Procedural Call

Remote Procedure Call (RPC) transparently extends method invocation to a distributed context, which can readily be applied to object oriented programming. However, the synchronous nature of RPC implies a strong time synchronization (on the consumer side) and also a tight space coupling, since an invoking object holds a remote reference to each of its invokees [8].

Object-oriented middleware (OOM) evolves and extends RPC by adding object-oriented concepts to it, such as inheritance, object references and exceptions [19]. OOM generally supports synchronous communication, although some solutions, for instance CORBA 3.0, supports both asynchronous and asynchronous communication.

Message-oriented Middleware

Message-oriented Middleware (MOM) systems provide distributed communication using an asynchronous interaction model. The interaction among communication partners is performed by means of messages, which allows for a loose coupling between participants; different systems based on different underlying technologies can be transparently integrated by relying on a common definition of message interaction and format. Message-oriented middleware is particularly well suited for distributed event notification and publish-subscribe systems. A messaging system allows for a number of additional features including fault tolerance, load balancing and priority schemes. Compared with RPC, scalability is greatly improved with MOM given the possibility of time and space decoupling.

Client-server systems

The client-server architecture model separates concerns (tasks) between communication partners. Servers, generally thick servers, provide services to, generally thin clients, that request (consume) services. In this sense, servers share their resources with clients and not otherwise. While this separation of concerns between clients and servers centralizes the main processing on the server, it alleviates the load on the client side, thereby allowing devices of heterogeneous capabilities to communicate with the server.

Peer-to-peer systems

In Peer-to-peer (P2P) systems the network nodes act as both service consumers and service provider [14]. This allows for a complete decentralized and distributed system where all nodes have equal or similar roles, for instance, in sharing their resources with other nodes. The main advantage of such model is that it deals naturally with scalability. In fact, the more nodes participates, the more resources, i.e., bandwidth, storage space, and computing power, are made available over the network, thereby increasing the overall system capacity. Both synchronous and asynchronous communication can be used in P2P. In [21], for instance, a publish/subscribe system is designed by means of Peer-to-peer interactions.

Publish/Subscribe

In the publish/subscribe interaction paradigm, subscribers register their interest in a specific event or a pattern of events with that publisher. The publisher notifies its listeners in case a matching event occurs. The publish/subscribe interaction model offers a complete decoupling in space, time, and synchronization [8]. Events are published by means of an event service that is responsible for notifying the respective subscribers. Therefore, publishers need not hold references to the subscribers or vice-versa. Furthermore, the use of an intermediate event service naturally allows for interacting parties to be notified asynchronously.

Service-oriented interaction

Service-Oriented Architectures (SOA) integrates heterogeneous applications in a distributed environment. It enables software systems to provide services to other applications through published and discoverable interfaces [3]. In the basic SOA model, service providers publish machine readable descriptions of their services in a public accessible registry; service requesters discover those services by querying the registry, and bind to the selected service dynamically. Automated service discovery, selection and binding are native capabilities of the SOA middleware. A direct benefit is loose coupling of services which eases the reconfigurability of service composition when the replacement of one or more service components is required.

3.1 Interaction Model for Opportunistic Networking

We now asses the suitability of the interaction models we have presented for their use in general opportunistic network environments.

In order for OppNets applications to properly *scale*, we opt for a message-oriented publish-subscribe model. This combination achieves a time, space, and synchronization decoupling, while messages can be efficiently routed to multiple subscribers, using message queuing and message caching mechanisms. In addition, to achieve scalability under the presence of heterogeneous devices, we include a client-server model as it naturally allows for the separation of concerns and tasks between powerful (server) and deprived (client) devices.

A publish/subscribe model implies the use of asynchronous communication. This is paramount for environments with *resource* limited devices, such as battery-powered devices. Since these devices will often employ energy saving mechanisms by switching off their systems, connections will be predominantly intermittent and the communication must be performed asynchronously. Message-oriented middleware (MOM) is a natural choice for asynchronous communication by exchanging messages.

In order to improve predictability of the execution time, synchronous processing shall be used. Even though the network is assumed to be predominantly delay tolerant, bounded time is an important factor in the assessment of opportunities and thus for *reconfigurability*. In the application interaction level, the Service-oriented architecture (SOA) is fundamentally important for the reconfiguration and composition of services relying on heterogeneous technologies.

Up to now, we have compared selected interaction models for each level in Figure 2. The above reasoning leaves us two choices for the Service-oriented architecture, for which two fundamentally different approaches are currently available [18]: Web service related technologies often abbreviated to WS-*, and Representational State Transfer (REST) that has been recently considered as an alternative to WS-* in the SOA domain. The SOA interaction model is important enough for reconfiguration among heterogeneous services to justify an in depth comparison between the two approaches. In the next section we introduce and compare them in view of our requirements for OppNets.

4 RESTful vs WS-*

Web services have been introduced to cope with limitations present in conventional middleware platforms. This is accomplished by relying on the service-oriented architecture (SOA), redesign of middleware protocols, and most importantly standardization [1]. There exist several formal definitions for the term *Web services* that often vary from very generic to very specific ones. In this work, we adopt the broad definition provided by the UDDI Consortium (W3C) which states:

> *"Self-contained, modular business applications that have open, Internet-oriented, standards-based interfaces. [20]"*

Up to the present moment, two main sets of technologies and concepts exist to be used in the integration of Web services: the WS-* and RESTful [18]. In the remainder of this section, we introduce and compare these technologies in view of our focus on opportunistic networks.

4.1 WS-*

The core of WS-* is divided in three main parts: (i) the Simple Object Access Protocol (SOAP), (ii) the Web Services Description Language (WSDL), and (iii) the Universal Description Discovery and Integration (UDDI). Based on the description available in [1], we briefly explain each category as follows.

SOAP: It provides a standardized application protocol that allows for loosely-coupled interactions between different Web services. It runs on top of other Internet protocols such as HTTP and SMTP and specifies how the information must be structured and exchanged between two parties. SOAP establishes a common message format based on XML and rules of how messages must be interpreted and processed. The XML description of the interaction between the communicating Web services and

can either follow the *Document* or *RPC* interaction styles. In the Document style, the body of the SOAP message is specified by an XML Schema and it does not need to follow specific SOAP conventions. Differently, in the RPC style, the structure of the message body needs to comply with the rules specified in detail by the SOAP specification.

WSDL: The WSDL is an XML document which describes a service, in particular, by specifying its interface. This is similar to Interface Definition Language (IDL) documents defined by conventional middleware. The WSDL document is divided into two parts: the *abstract* and *concrete* parts. The former specifies the operations that can be requested to the service included in *port types*, the *messages* that can be exchanged between the two parties, and a vocabulary for the data types used by the service. The latter manages the description of the service address, the underlying Internet protocol running below SOAP, and finally, the interaction paradigm.

UDDI: It is a discovery repository which is analogous to a "telephone directory" of Web services. Among its components, the *businessEntity* describes the organization that provides the Web services; the *businessService* describes the services and their classification; and the *bindingTemplate* describes the technical information of a particular service. In particular, the latter includes references to documents called *tModels (technical models)*. *tModels* contain a generic description of any type of specification and are used, e.g., to point to the actual service's WSDL.

From a practical point of view, the functioning of the interaction between two parties is done as illustrated in Figure 3. An organization requiring a certain service may first perform a search in a well-known UDDI service. When a service with the required specification is found, the organization retrieves the WSDL containing the necessary information of how the interaction can be done with that particular service. Finally, a Web service client is built based on the WSDL retrieved specification. The web service is able to interact with the Web service provider by exchanging SOAP messages, which are encapsulated into, for instance, HTTP messages. Upon the receipt of such SOAP messages, a translation is made on the service provider that may yield into invocation of existing back-end systems, for instance, using on CORBA [16].

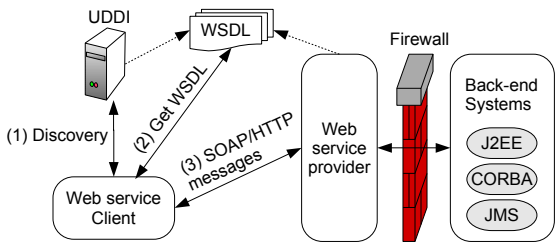

Figure 3: WS-* Overview

4.2 RESTful

The Representational State Transfer (REST) is an architecture style proposed in [9] which aims to meet desired properties of a modern Web architecture. REST addresses aspects such as portability, scalability, visibility, and reliability, by incrementally adding constraints to the *null style*, i.e., an architecture model without any sort of constraint. These characteristics are defined in [12] and listed as requirements for the proper interaction between Web applications, as follows:

Client-server style: The separation of concerns between clients and servers helps improving portability and scalability by allowing each side to evolve independently.

Stateless: Connections between clients and servers must be done statelessly in the sense that the state of sessions is only stored at the client side. This allows for: (i) visibility, as a single client request will contain all necessary information for the server to understand and execute it; (ii) improvement in reliability as the recovery of partial failures can be easily performed; and finally (iii) scalability is enhanced, as servers do not need to store any state information of a client request. However, this architectural decision comes with the disadvantage of increasing the communication overhead due to necessity of repetitive data in every client request.

Caching: In order to improve network efficiency, the *cache* constraint is introduced. This means that every response from the server should contain a label stating whether the information is cacheable or not. All cacheable information is stored at the client side for future use in such a way that the number of requests is reduced.

Uniform Interface: It defines a common interface for every application. The main advantage is that by increasing generality and simplifying the system, implementations are fully decoupled from the services they provide. On the other hand, a uniform interface might affect efficiency, since it might not always match directly the application's needs.

Layered system: The scalability is further improved by the use of layered system constraints. Layered systems decrease the complexity of individual components by restricting the knowledge they need to manage in a single layer. Together with lower complexity, they also permit an independence between layers and how they evolve.

Code-on-demand: It allows clients to extend their functionality by downloading on-demand new features from scripts or applets.

As an architecture style, REST can be applied to different existing technologies. Technologies applying REST principles are often referred to as *RESTful*. A RESTful Web service is often referred to as service implemented with an HTTP interface and following the REST principles. According to these principles, any information is

viewed as *resource* which can take form of documents, images, a certain service, a collection of resources, or even abstract concepts. These resources are retrieved by means of *resource identifiers*. By following these principles, a RESTful Web service is simply a collection of resources where each collection and its resources are identified by Uniform Resource Identifiers (URIs). The representation of resources can take any form specified by the Multipurpose Internet Mail Extensions (MIME) [10] standard, such as XML and JSON (JavaScript Object Notation). The uniform interface of a resource takes the form of the HTTP methods, namely POST, GET, PUT, DELETE.

Table 1: Usage of HTTP methods in RESTful

Resource	***Collection***: http://eg.com/service/	***Resource member***: http://ex.com/service/r1
GET	Return the URI list of all collection members.	Return the representation of the resource.
PUT	Replace the entire collection with another one.	Update a specific member of the collection.
POST	Create a new entry in the collection (ID automatically generated).	Create a new entry given a specific ID.
DELETE	Delete the entire collection.	Delete a specific member of the collection.

Table 1 exemplifies how HTTP methods can be utilized over RESTful service URIs. For each method, there is a direct mapping for the operation to be executed over the target resource. In this way, the operations over resources become predictable and simple. In addition, complex requests can be easily constructed by inserting additional parameters to, for instance, a GET HTTP request: *http://example.com/users/User?firstName=Frank&lastName=Schut*

Differently from WS-*, there is no standard for defining interfaces in RESTful Web services. Either the Web Application Description Language (WADL) or WSDL 2.0 can be used. Nevertheless, in most cases an interface definition may not even be necessary, since all operations available by the RESTful Web service are known beforehand due to the use of a uniform interface. Furthermore, a simple informal description of the interface semantics explaining how the interface must be used may already suffice for the integration between different services. Similarly, there is no standardized manner for discovering services with RESTful. A straight-forward manner is to establish the integration of different services manually by directly contacting a certain Web service provider. Another approach is to crawl over sub-links of a given starting-point URL and navigate into sub-resources.

4.3 Comparison between RESTful and WS-*

Several comparisons between RESTful and WS-* have been presented in the literature. They are either based on architectural principles and decisions [18]; general advantages and disadvantages for Web services choreography [22]; or level of loose coupling [17]. Differently, in this work we take into consideration *scalability*, *resource efficiency*, and *reconfigurability*, as outlined as requirements in Section 2.

When considering Web intermediates such as caches and gateways, RESTful services can notably *scale* better than WS-* services [4]. With RESTful services, every HTTP request is transparently processed and understood by intermediate cache servers along the network path. On the other hand, intermediate cache and proxy servers might not have the required intelligence to process WS-* SOAP messages over HTTP, which are mainly encapsulated into HTTP POST messages. This may lead to a lower performance in terms of latency and scalability.

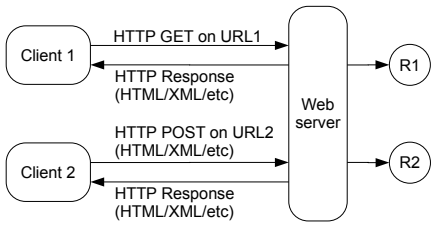

Figure 4: Request processing with RESTful

RESTful Web services are generally classified as lightweight in terms of *resource utilization* in comparison with WS-* [4]. The main reason is the additional processing overhead introduced when exchanging messages with WS-*. As illustrated in Figure 4, because of the direct mapping of HTTP commands to the operations over the resources present in the server, a simple HTTP command over the corresponding resource URI suffices for the Web server to understand and perform the request.

Differently, WS-* SOAP requests are always encapsulated into HTTP POST requests. In order for these requests to be processed, an additional SOAP server is required, as shown in Figure 5. This is the result of using HTTP as a transport protocol rather than a application protocol.

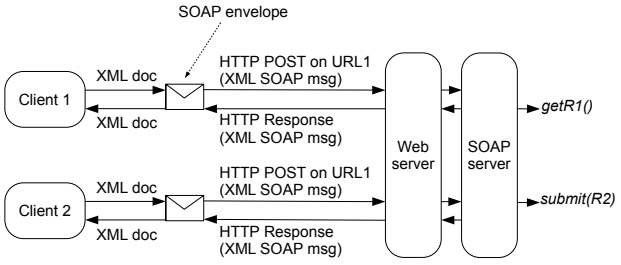

Figure 5: Request processing with WS-*

Furthermore, since operations over resources with RESTful Web services are done by directly invoking HTTP methods, the size of request messages are generally smaller than SOAP messages. Furthermore, RESTful resources can be represented with various formats, mostly XML or JSON, whereas SOAP messages are strictly represented in XML. JSON is based on a subset of the

JavaScript Programming Language and has been designed to be a lightweight data-interchange format [5].

In terms of *reconfigurability*, RESTful Web services rely on well-known W3C/IETF standards, e.g., HTTP, XML, URI, MIME and require minimum tooling to be deployed [18]. For their basic functioning, they only need HTTP clients and servers, which in fact have already become pervasive and available for all major programming languages and mobile operating system/hardware platforms. Furthermore, thanks to URIs and hyperlinks, RESTful has shown that it is possible to discover Web resources without an approach based on compulsory registration to a (centralized) repository. The uniform interface used by RESTful services eases the immediate integration between services by making their interfaces predictable. In fact, mobile devices in an ad-hoc communication can query other devices for additional service or resource by simply performing a HTTP GET operation. Differently, WS-* services use SOAP messages and XML descriptions that differ from service to service. Although the use of a uniform interface has been criticized for the difficulties sometimes found when mapping Web service methods to HTTP verbs in RESTful Web services, we argue that an uniform interface is more suitable for an ad-hoc integration between limited mobile devices.

Overall, RESTful characteristics comply with our requirements and are better suited for opportunistic networking when compared with WS-*.

5 RESTful Application

In this section, we exemplify an infrastructure-less opportunistic network with a combined publish/subscribe early-warning and monitoring system for wildfire. The diagram shown in Figure 6 provides an overview. Sensor devices are spread in a *monitoring* hayfield. Along the field is a daily public *transport* (bus shift) to an urban area, where it can get access to the Internet backbone (*alarm center*). Sensor devices in the field harvest their energy from light and heat.

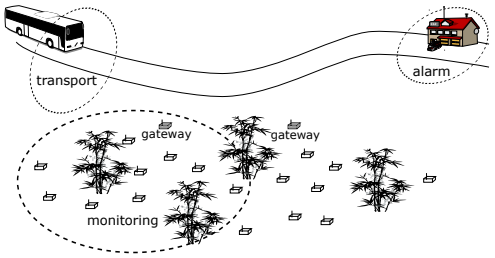

Figure 6: Wildfire Scenario

We rely on a RESTful communication by setting URIs with a uniform interface to each node. Although there exist light implementations of the full Internet stack, i.e., HTTP, TCP, IP, for tiny nodes [7], we assume that any specific protocol following the REST principles can be deployed in the sensor nodes. The remaining entities, namely the buses and the alarm center, rely on HTTP for communication. One example of a RESTful publish/subscribe system can be found in [13]. The gateways residing near the road serve as translators between the sensor island and the external world, similar to the architecture proposed in [15].

Table 2: URI locator

op	uri
PUT	`/election/broadcast`
PUT	`/election/capabilities`
PUT/GET	`/mode`
GET	`/temperature`
GET	`/alert`
PUT	`/monitor/temperature`
PUT	`/monitor/alert`
PUT	`/route`

Sensor nodes may operate in one of the three modes: sensing, monitoring, and hibernating. Since a fire event cannot be derived deterministically from a single node's observations, a monitoring node will combine multiple neighboring observations on which it decides to raise the alert level or not. Potential alerts are verified by waking up inactive nodes and reading their sensors. Nodes hibernate as much as possible since their energy capacity is strictly budgeted per day. Although in case of a fire, the heat will give them access to abundant energy but they will die in the fire. Henceforth, frequent reconfigurations are required even during normal operation. Clusters are formed dynamically to measure, collect and interpret data in the most energy efficient way. In normal situations, log data migrates towards the road for the bus to pick up and to deliver at the backbone (alarm center).

To illustrate the RESTful interaction model for these sensor nodes, consider them to implement the set of URIs listed in Table 2. The protocol involves three phases. Phase 1, is the election and configuration phase in which sensor nodes join clusters with monitors, sensors and hibernating nodes. Hereto, nodes broadcast their availability to participate in the measurement (put) `uri:///election/broadcast`, and responses with the capability of each node are put in `uri:///election/capabilities`. With this information, nodes decide autonomously on their preferred mode of operation, which is multicast through `uri:///mode`. Phase 1 is concluded with the configuration of a cluster, i.e., nodes subscribe to alert messages through `uri:///monitor/alert`. Phase 2 is the measurement phase. Monitor nodes get readings from sensor nodes through `uri:///temperature` and, occasionally, route their log data to the alarm center through `uri:///route`. This involves an intermediate communication with gateways (translators) and buses. At frequent intervals, the network restarts the election phase

to re-evaluate its configuration. In case of an anomaly, the system enters Phase 3, the alert phase. This phase triggers monitors to read the temperature and alert states of the known nodes through `uri:///temperature` and `uri:///alert` respectively. Further, the publish mechanism of `uri:///monitor/alert` invokes a wakeup call through `uri:///monitor/temperature`. In case of a real fire, all reachable nodes will wake up and participate in the routing of the alarm data. In case of a false alarm, the system returns to the measurement state or resets to the election phase.

6 Conclusion

In this paper, we have evaluated various application interaction models for Opportunistic Networking (OppNet). First, we have derived fundamental requirements based on a general OppNet scenario. Given these requirements, we have casted a classification of main interaction models and compared them in view of opportunistic environments. In particular, we have payed special attention to the comparison between RESTful and WS-*, since they present two fundamentally different approaches for service-oriented architectures. Based on our analysis, we conclude that a publish/subscribe model with RESTful interfaces provides efficient means to achieve a scalable, resource efficient, and highly reconfigurable distributed application. This is in part reasoned by the predominant use of asynchronous communication, as it suits the need for devices to perform energy saving mechanisms. Other components include a client-server network architecture to allow for the separation of concerns between powerful and limited devices, and synchronous processing in a lower level of interaction model. The latter provides a more accurate deterministic prediction of execution time and shall be employed to improve the assessment of opportunities. Finally, we have presented a basic application showing how the REST concept can be used in opportunistic networks. In future work, our goal is to evaluate the performance of the proposed combination of interaction models and to model applications that exploit opportunistic networking.

Acknowledgments

The authors express their sincere gratitude to the ARTEMIS iLAND Project partners for their fruitful discussions on interaction models for distributed applications. In particular, we thank TI-WMC for their suggestions in the early-warning wildfire application scenarios.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web services: concepts, architectures and applications*. Springer Verlag, 2004.

[2] C. Boldrini, M. Conti, and A. Passarella. Context and resource awareness in opportunistic network data dissemination. *International Symposium on a World of Wireless, Mobile and Multimedia Networks*, (027918):1–6, June 2008.

[3] K. Channabasavaiah, K. Holley, and E. Tuggle. Migrating to a service-oriented architecture. *IBM DeveloperWorks*, 16, 2003.

[4] R. Costello. REST - Representational State Transfer [online]. *http://www.xfront.com/REST.html*, 2002.

[5] D. Crockford. JSON: The fat-free alternative to XML. In *Proc. of XML*, Boston, USA, 2006.

[6] K. De Zoysa, C. Keppitiyagama, G. Seneviratne, and W. Shihan. A public transport system based sensor network for road surface condition monitoring. In *Proceedings of the workshop on Networked systems for developing regions*. ACM New York, NY, USA, 2007.

[7] A. Dunkels, T. Voigt, and J. Alonso. Making TCP/IP viable for wireless sensor networks. In *Proceedings of the First European Workshop on Wireless Sensor Networks, work-in-progress session, Berlin, Germany*, 2004.

[8] P. T. Eugster, P. A. Felber, R. Guerraoui, and A. M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):131, 2003.

[9] R. T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, 2000.

[10] M. N. Freed. MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for specifying and describing the format of Internet message bodies. *RFC 1521, Bellcore, Innosoft*, 1993.

[11] R. Friedman, D. Gavidia, L. Rodrigues, A. C. Viana, and S. Voulgaris. Gossiping on MANETs:the beauty and the beast. *ACM SIGOPS Operating Systems Review*, 41(5), 2007.

[12] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of software engineering*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 2002.

[13] P. Hintjens. RestMS - a RESTful Messaging Service [online]. *http://restms.org/*, 2009.

[14] M. O. Junginger and Y. Lee. *Peer-to-Peer Middleware*, chapter 4, pages 81–108. Wiley, 2004.

[15] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim. TinyREST: A protocol for integrating sensor networks into the internet. In *Proc. of REALWSN*, 2005.

[16] T. J. Mowbray and R. C. Malveau. *CORBA design patterns*. John Wiley & Sons, Inc. New York, NY, USA, 1997.

[17] C. Pautasso and E. Wilde. Why is the web loosely coupled?: a multi-faceted metric for service design. In *Proc. of the 18th international conference on World wide web*, pages 911–920. ACM New York, NY, USA, 2009.

[18] C. Pautasso, O. Zimmermann, and F. Leymann. Restful web services vs. "big"' web services: making the right architectural decision. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 805–814, New York, NY, USA, 2008. ACM.

[19] H. Pinus. Middleware: Past and present a comparison, 2004.

[20] UDDI Consortium. UDDI Executive White Paper [online]. *http://uddi.org/*, 2001.

[21] D. Wang and J. Liu. Peer-to-Peer asynchronous video streaming using Skip List. In *Proc. of ICME*, 2006.

[22] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing web services choreography standards—the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29, 2005.