# Sensor network integration by means of a Virtual Private Network protocol

D. Villa, F. Moya, F.J. Villanueva, Ó. Aceña, and J.C. López

School of Computer Science, University of Castilla-La Mancha
Paseo Universidad 4, 13071 Ciudad Real, Spain,
david.villa@uclm.es
WWW home page: http://arco.esi.uclm.es

**Abstract.** Sensor networks are becoming an essential part of smart environments. However, most previous systems rely on ad-hoc mechanisms to access the sensor network from the enterprise information system. We propose the use of object oriented middlewares to provide a Virtual Private Network in which all involved elements (sensor nodes or computer applications) will able to communicate as if they all were in a single uniform network.

**Keywords:** network integration, virtual networks, sensor networks, smart objects

## 1 Introduction

Integration of sensor networks[1] in enterprise information systems is still troublesome. The interaction among applications and sensor services usually requires specific mechanisms often centralized on application-level gateways. Sensor data comes through a single node with two interfaces, one for the sensor data (e.g 802.15.4 wireless interface) and another for the enterprise information network (e.g Ethernet). In many situations this gateway runs custom software to adapt the protocol stack from the sensor domain to the enterprise domain. However this kind of infrastructure implies a single point of failure. From an engineering point of view, a single gateway implies a single point of failure, attending to specific application, it could make difficult and inefficient network topology, etc. i.e. the network topology between domains could be arbitrary and not dependent on integration logical infrastructure.

At logical level, it is inconvenient for actor/actuator nodes and complicates, or just makes impossible, free interaction among applications and sensor nodes and particularly interaction among sensor nodes.

[1] This research was supported by the Spanish Ministry of Science and Innovation and CDTI through projects DREAMS (TEC2011-28666-C04-03), ENERGOS (CEN-20091048), PROMETEO (CEN-20101010), by the Regional Government of Castilla-La Mancha and ERDF under project SAND (PEII11-0227-0070), and by Cátedra Indra UCLM.

This paper introduces a novel approach to achieve a more flexible and decoupled way to provide and request sensor services supporting several gateways among sensor and enterprise domains (sometimes called *multi-sink*). By means of a common application-level protocol, sensor nodes and applications can interact in any scenario, even among sensor nodes belonging to remote networks. Instead of designing a new application protocol from scratch, we focus our attention in the protocols used by object-oriented middlewares (i.e CORBA or ZeroC Ice). These middlewares has been traditionally used in scalable and efficient distributed heterogeneous applications so we start with well-known and tested protocols. The application protocol used in these middlewares really marshall/unmarshall invocation messages between distributed objects. We already integrate these type of protocols in wireless sensor networks [9] but always inside of the same sensor network domain. Of course, that allows to use an object oriented middleware, which transforms the sensor network integration in a case of distributed heterogeneous programming.

## 2 Related Work

In a way [1] aims at similar goals: "Unlike application-level gateway, that require semantic knowledge of each application in order to make a routing decision, the overlay gateway routes based on sensor network layer information". They propose an *overlay network* to interconnect applications with sensor nodes extending the sensor network internal protocol over Internet. It works as a *virtual sensor network* thanks to *overlay gateways*. In their words: "It is a sensor network overlaying IP". The gateway encapsulates the sensor network protocol packets (including network, transport and application headers) on TCP or UDP segments. As the sensor network stack is preserved, components at hosts (virtual sensors) need to process all of these *strange* headers at the application layer in order to maintain the illusion of a single flat network.

SenseWrap [3] takes the other way. They refer to *virtual sensors* as the wrapped versions of the actual sensors. They focus on self-configuration providing standard Zeroconf to discover and find sensor services. It is a middleware to get *IP overlaying to the sensor network*. SensorWrap uses a single application-level gateway, the model that we try to avoid. Tenet [10] is a more sophisticated network architecture that divides the sensor network in a set of tiers. Each tier has a master and several nodes. Most of processing and application specific tasks run in the masters. Hence it is a multi gateway approach that avoids a single point of failure but may significantly degrade the network performance if some of the masters fail.

The "all over IP" approach could solve the problem but introduces overhead even in low-footprint implementations (i.e uIP [2]) and may not be affordable for some sensor domains. Other protocols like Message Queue Telemetry Transport protocol [5] (MQTT) from IBM are for telemetry applications so it does not support actuators and individual sensor-to-actuator interactions. Our approach uses the network and transport protocol stack most appropriate at each domain,

only at application level we use a common protocol. As far as we know, none of existent approaches provide an integration mechanism with such a flexibility and transparency.

Network interconnection (among sensors, trunk networks or whatever) is feasible when both use the same protocol, at least at network layer. However in practice there are many scenarios and applications which impose proprietary or non-compatible protocols in the sensor network. There are many reasons (i.e energy efficiency, real time characteristics, etc.) but we will not analyze them here. In these circumstances application-level gateways are used. Our concept is quite similar to the conventional *virtual private network*. That is: *hosts* suitable to interchange information have identifiers in the same address space, creating the illusion that they are all neighbors although part of them are remotely connected through other networks. Note that our work is *not* related with the Virtual Sensor Network (VSN) concept. VSN [6] is a mechanism to select (sometimes dynamically) a subset of sensors and provide them to the user/task as a different (virtual) network, so that the focus is on providing sensor node logical aggregation. Neither to be confused with VPSN (Virtual Private Sensor Network [2]) which is *virtual* in other sense. It provides a per-user sensor network vision. Neither of them are related to network interconnection.

## 3 UVPN for sensor networks

UVPN (Ubiquitous Virtual Private Network) uses the same VPN concept available in TCP/IP networks although implemented at a higher abstraction layer. Each *host* may have one or more *object adapters*. An object adapter is responsible to expose local objects to the network. Each object adapter is accessible through *endpoints*, i.e. logical network connection points. Emulating the conventional VPN model, we assign homogeneous addresses to all involved components, regardless of whether they are physical sensor nodes or PC applications. To achieve this, we need a new kind of endpoint (*UVPN endpoint*). The UVPN endpoint, as the conventional VPN counterpart, uses the same underlying transports, ie. TCP, UDP or SSL in the Internet case. We are using ZeroC Ice [4] in our current prototypes, although any other object-oriented middleware could be used instead. Sensor nodes with minimal footprint are able to process application messages using the underlying protocol [9]. Of course, those virtual logic addresses need to be mapped to the corresponding underlying address (equivalent to neighbor discovery in conventional protocols). Because this kind of translation may be expensive and complex for a sensor network, our approach here uses identifiers which may be directly mapped to the node physical addresses. UVPN endpoints encapsulate their own communication details. An example of proxy to a remote sensor object using the UVPN endpoint would be "OBJ2 -d:uvpn -h 0x01" where "0x01" is the *native* sensor node address. These virtual nodes may hold sensors and actuators fully indistinguishable from their actual

[2] GENI Project `http://www.geni.net/`

counterparts. Contrary to [1] we do not need a common network layer protocol, just a common application layer protocol.

The main component of the UVPN architecture is the *UVPN switch*, a service that should reside in a host having physical interfaces to both networks: sensor and trunk. Its goal is similar to a conventional *VPN switch*. The switch is aware of any compatible remote object adapter, those which have endpoints supporting UVPN addressing. The *switch* is a conventional distributed object implementing the `UVPN::Switch` interface shown below:

```
module UVPN {
  class Address {};

  interface Transceiver {
    void send(Address addr, ByteSeq payload);
  };

  interface Switch extends Transceiver {
    void add(Address addr, Transceiver* prx);
    void remove(Address addr);
    Transceiver* find(Address addr);
  };
};
```

For better transparency the UVPN endpoint (in a the conventional computer side) performs the registration on behalf of the adapter. When an endpoint is instantiated, it invokes the `Switch.add()` method in the designated remote switch to bind a sensor network address (`addr`) to a callback object provided by the UVPN endpoint. Later, the switch can resolve remote *virtual sensor nodes* using these associations between addresses and endpoints. This mechanism is functionally equivalent to creation of a tunnel in an conventional VPN.

### 3.1 Use cases

This section discusses the different communication scenarios among node services and applications, or among nodes themselves.

*Application to node.* In the simplest scenario, a client application requests the status of a remote sensor. In the application-level bridge-based approaches [7, 11, 8], the sensor network specific protocols store last measured sensor values in the bridge. Later the client explicitly query the bridge giving some sensor identifier. That was the cause of many important problems: bridge complexity, lack of node autonomy, single point of failure, etc.

With UVPN, the client (in the trunk network) performs a conventional remote object invocation on a proxy representing the remote sensor. Under the scene, the UVPN endpoint knows (by configuration) where is the *switch* and invokes `Transceiver.send()` passing the whole client invocation message as argument. The switch receives the message and tries to find (by means of the `Switch.find()` method) a *transceiver* (virtual node) for the node address specified as the first argument in the `send()` invocation. In this case, the address is not found, the message is sent to the sensor network physical interface, directly connected to the computer running the switch service. The message goes to *the air* and should be received by the target node.

*Node to application.* UVPN allows sensor nodes to behave as clients, that is, nodes can transparently invoke remote objects in the trunk network. That is a very rare feature in sensor network middlewares which was previously handled with ad-hoc non-generic solutions.

In this case, the sensor node just sends a conventional invocation preceded by the destination node address. The switch receives the message through the radio interface and looks for the destination address (with `Switch.find()`). In this scenario, the method returns a virtual node proxy. The switch uses it to forward the invocation to the computer. The UVPN endpoint in the computer application receives the message and gives it to the object adapter. Finally, the corresponding servant method is executed.

*Node to neighbor node.* Any node may send method invocations to any other neighbor in the same physical network using exactly the same mechanism described in the previous section. This means the invocation mechanism is location transparent, i.e. the is not aware of the exact location of the destination object (sensor node or computer application).

In this case, the switch will not find a remote virtual node and it will send the message to the radio interface again. This is also useful when the sensor network does not implement multi-hop routing because the switch will forward messages automatically. If destination nodes receive replicated messages through different paths they are automatically discarded by just checking the sequence number in the header.

*Node to non-neighbor node.* There is a more interesting use case, also very rare in previous works. Two or more distant sensor networks (with their respective UVPN switches) connected to the same trunk network (and this network may be Internet). In this situation, a sensor node may invoke other remote sensor node (in a different network) using switches to forward the message towards the trunk network. As explained before, this requires that the local switch knows whether the remote object is accessible through itself. It implies the registration of all sensor nodes in a central switch (the *the root switch*). Obviously, were are talking about a hierarchical switching protocol. Local switches have a fallback switch (a *default path*) that knows where is each sensor node. Figure 1 illustrates the described scenario.

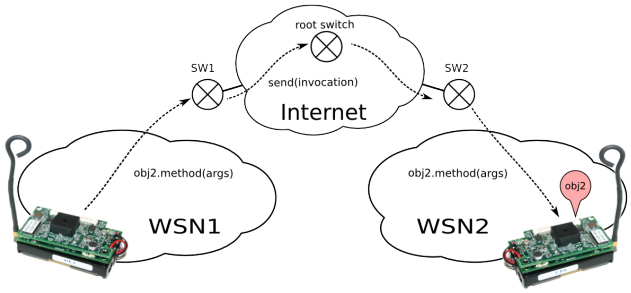


**Fig. 1.** Invocation from a sensor node to a remote node using UVPN

Figure 1 makes clear that UVPN works like a tunneling protocol although it is built on the application layer. The switch operation is a bit more complex

when more that one sensor network participates in the communication. When a sensor node (as client) wants to send a message to another node, it builds the message as if destination node were a neighbor, although it is in different physical network. The local switch will receive the message, and check whether the destination address is registered on its table. If so, it sends the message using the associated *Transceiver*. Otherwise, it will use the fallback switch to send the message (using the method `Transceiver.send()`).

When this fallback switch (or root switch as stated above) receives a message, it will check if destination was already registered. If so, it sends the message again, using the matching transceiver. If destination is unknown, then it will deliver the message to the remaining transceivers configured, one on each port (flooding). Only discards the arriving gate (in order to avoid loops).

When the last switch receives the message, it will check again its table. If it is registered as a virtual node, it will send the message using the given transceiver. If not, the message will be sent to the other interfaces (i.e. the radio interface, or other registered switchs, as well as are different from the arrival one).

*Application to application?* It is also possible to communicate no-sensor clients and objects using UVPN. However, given that the middleware supports several transports at same time, it is more convenient to use TCP/IP endpoints. Without any loss of generality this means that UVPN is used only when needed, i.e. only when a sensor node is involved either as client or object.

## 4 Simulation and results

Several OMNet++ simulations are built to demonstrate the correct operation of UVPN. An example smart-grid application monitors the total electrical consumption of a house to avoid current overload. Each socket measures the current through it and is able to interrupt power supply to turn on/off the load connected to it.

Three *load states* are defined: **NORMAL** when the total consumption is less than 70%, **WARNING** when the total consumption is between 70% and 90%, and **CRITICAL** when the total consumption is greater than 90%.

When the user (up)plugs an appliance to a socket, the device itself sends a message to a monitor service (loadMonitor) running in a `server`. The loadMonitor decides the new state and notifies changes in an event channel. Subscribers of any kind (e.g. applications or sensors) are allowed since they all behave as normal distributed objects.

We will describe in detail the event sequence occurring in the simulation: 1) in the initial state (NORMAL) there are 4 unplugged sockets; 2) an appliance is plugged to the socket `node1`, which measures the current and sends a message to the loadMonitor service; 3) the loadMonitor service receives the message, estimates that system load exceeded the WARNING threshold, and sets load state to WARNING; 4) another appliance is plugged into socket `node2`, which sends a message notifying it; the loadMonitor detects that load is above CRITICAL threshold, so it sets load state to CRITICAL. The sockets (they are channel

subscribers) receive the message; those of them without any device plugged in immediately cut the line and turn on a red LED to visually warn the users and preventing overload condition; 5) when some of the appliances are unplugged (or turned off) the loadMonitor can set load state to lower states allowing new appliances to be plugged or activated.

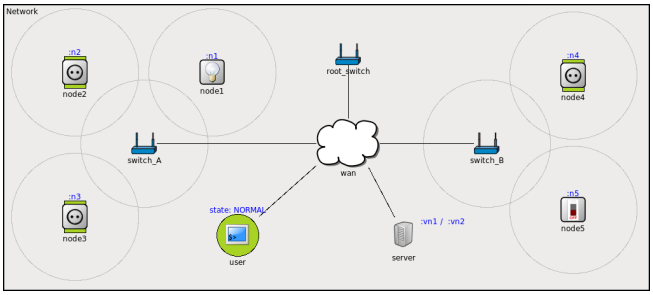


**Fig. 2.** UVPN simulation: direct remote communication among sensor nodes in distant networks

This example illustrates UVPN direct and duplex communication among sensor/actuator nodes and application objects running on conventional computers in the trunk network. Communication may be initiated by any party and they all may act as clients or objects. Besides, sensor nodes may act as publishers or subscribers of event channels.

There is a more complete simulation involving 2 sensor networks connected to the same IP network through their corresponding UVPN switches (see Figure 2). It represents the same smart-grid application introducing bulbs and electrical switches that are new electrical loads. To illustrate the communication among remote sensor/actuator nodes, the switch *node5* send messages *set()* true/false to turn on/off the bulb `node1`. As in the previous case, nodes 1-4 send load messages to the loadMonitor service and receive load state messages from the event channel. There are other nodes (e.g. `user`) which receive notifications from the loadMonitor service. In this case, `user` will alert people about an overload.

The simulations source code, brief documentation and screencasts are available for download at `http://arco.esi.uclm.es/uvpn`. UVPN has been implemented with the ZeroC middleware in a demonstration kit called *Motebox*. This kit is used to show different features on access and interconnection among PC applications, middleware services and sensor nodes. More information may be found at `http://arco.esi.uclm.es/motebox`.

## 5 Conclusions and future work

It is interesting to analyze the differences in relation to [1]. It encapsulate the whole sensor network protocol stack, thus requiring specific support for each

sensor network protocol stack at *virtual nodes*. With UVPN, it does not matter which protocol stack is used as long as all peers use the same inter-ORB protocol (at application layer) and the same addressing scheme, that is, the typical requirements of a inter-network protocol. Furthermore the middleware also provides valuable common services such as object persistence, indirect binding, location transparency, server deployment and many other advanced features.

As far as we know UVPN is the first solution able to communicate sensor or actuator nodes among themselves and with software objects running on conventional computers in a transparent way, without application specific delegates, application bridges or ad-hoc protocols. However, there is a constraint: all sensor nodes must use the same physical addressing scheme. As ongoing work we are interested in solving that limitation by generalizing the UVPN approach using a global addressing scheme (see section 3) and providing homogeneous dynamic routing mechanisms through massively heterogeneous networks.

## References


1. H. Dai and R. Han. Unifying micro sensor networks with the internet via overlay networking. In *Intl. Conf. on Local Computer Networks, LCN'04'*, 2004.
2. A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of WWIC 2004*, Frankfurt (Oder), Germany, February 2004.
3. P. Evensen and H. Meling. SenseWrap: A service oriented middleware with sensor virtualization and self-configuration. In *Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 261–266, December 2009.
4. M. Henning and M. Spruiell. *Distributed Programming with Ice*. ZeroC Inc., May 2008. Revision 3.3.0.
5. U. Hunkeler, H.L. Truong, and A. Stanford-Clark. MQTT-S - A publish/subscribe protocol for wireless sensor networks. In *COMSWARE*, pages 791–798. IEEE, 2008.
6. A.P. Jayasumana, Q. Han, and T. Illangasekare. Virtual sensor networks a resource efficient approach for concurrent applications. In *In International Conference on Information Technology: New Generations*, 2007.
7. P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, October 2002.
8. S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.
9. F. Moya, D. Villa, F.J. Villanueva, J. Barba, F. Rincón, and J.C. López. Embedding standard distributed object-oriented middlewares in wireless sensor networks. *Wireless Communications and Mobile Computing*, 9(3):335–345, 2009.
10. J. Paek, B. Greenstein, O. Gnawali, K.-Y. Jang, A. Joki, M. Vieira, J. Hicks, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. *ACM Transactions on Sensor Networks (TOSN)*, 6(4), 2010.
11. Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Record*, 31:2002, 2002.