Workshop Proceedings of the 8th International Conference on Intelligent Environments
J.A. Botía et al. (Eds.)
IOS Press, 2012
© 2012 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-61499-080-2-389

MidBlocks: A Supervising Middleware for Reliable Intelligent Environments

Rafael BAQUERO[a,1], José RODRIGUEZ[a], Sonia MENDOZA[a]
and Dominique DECOUCHANT[b]
[a] *Departamento de Computación, CINVESTAV, México*
[b] *Univarsidad Autónoma Metropolitana, Unidad Cuajimalpa, México*

Abstract. Intelligent Environments (IEs) are expected to have a dramatic impact on our daily lives in the near future. Environments that can assist users in their daily activities, aid the elderly in continuing productive and independent lives, keep people safe and help them make better use of resources can provide great benefits. However, due to the multidisciplinary nature of IEs, the complexity of the technologies involved, and the distinct requirements of different user groups, the development of full scale IEs is a difficult and error prone task. Although a robust design is a fundamental part in the development of an IE, proper supervision of the system for failure detection is important to guarantee the reliability of the systems. In this paper we propose MidBlocks, an event-based component supervising middleware. Through the use of MidBlocks, Intelligent Environments can perform constant supervision of their components and communication links in order to warn users in case of a system malfunction or to perform corrective actions such as dynamic system reconfiguration.

Keywords. Intelligent Environments, Middleware, Distributed Systems, Component Supervision

Introduction

Intelligent Environments are heterogeneous distributed sensor-actuator systems including multimedia presentation services, building automation and control components, intelligent physical objects, wireless sensor network nodes, nomadic personal or shared devices, and many other systems and entities [1]. It is expected that IEs will enrich the lives of humans by adapting the environment to the inhabitant's activities and by providing services such as assistance to users with special needs. In some cases, failure of the IE can have serious consequences to the inhabitants of the environment. Therefore a fundamental issue in the development of any IE is dependability. Dependability is the ability of a system to deliver service that can justifiably be trusted [2]. However complex systems can, and sometimes do, fail. Being able to detect system failures to warn users or to take corrective actions is an important aspect of any IE.

Due to their nature, Intelligent Environments are complex systems that require the integration of multiple components. A frequent solution employed to link these

[1] Corresponding Author: Rafael Baquero S., Depto. de Computación, CINVESTAV, Av. IPN 2508, San Pedro Zacatenco, México D.F., México, 07360. E-mail: rbaquero@computacion.cs.cinvestav.mx

components is the use of a middleware. A middleware is a communications mechanism through which the components of a distributed system can interact. To improve reliability, some systems such as industrial control systems employ the communications mechanism to simultaneously perform component supervision tasks.

In this paper we present MidBlocks, an event-based publish/subscribe middleware which provides a communication mechanism for distributed systems while simultaneously supervising the components of the system.

The rest of this paper is organized as follows. In section 1 we describe the basic operation of event-based publish/subscribe middleware. Next in section 2 we mention other work performed in the area of AmI systems dependability. Afterwards, in section 3, we describe the architecture and operation of MidBlocks. In section 4 we provide a brief description of FunBlocks, a modular, minimalist framework for the integration of AmI systems which uses MidBlocks. Finally in section 5 we describe some future work which we plan on carrying out on MidBlocks.

1. Event-Based Systems

A middleware is a software layer which allows the components of a distributed system to interact [3]. Many different types of middleware have been developed, each based on a different paradigm, and targeted towards the solution of a certain class of problems.

A common class of middleware is the event-based publish/subscribe model. In an event-based middleware entities exchange information through discrete packets called events [4]. An event represents any discrete transition that has occurred and is signaled from one entity to a number of other entities. In the publish/subscribe model producers publish the information they can generate on an event manager, while consumers subscribe to the information they want to receive. The middleware can further be provided with a store and forward entity. When a store and forward entity is used producers send events to the entity that stores them and forwards them to consumers. The use of an event-based, publish/subscribe middleware with a store and forward mechanism allows full space, time, and synchronization decoupling between event producers and consumers. This type of middleware is used in applications that have strong requirements in terms of reliability but do not need a high data throughput [5].

2. Dependability in Middleware and Ambient Intelligence Systems

Dependability in middleware and Ambient Intelligence systems has been a subject of interest in the last few years. In [6] Rice and Beresford explore build dependable pervasive computing applications which utilize a distributed middleware. The authors define a dependable system as one that either performs within specification, or provides suitable feedback to users when faults occur. They use accountability as a mechanism for constructing a dependable system, where accountable applications are capable of describing why a particular action was (or was not) taken.

In [7] Coronato presents Uranus, a middleware aimed at the development of Ambient Assisted Living and vital signs monitoring applications. The objective of Uranus is to provide a middleware infrastructure for the rapid prototyping of Ambient Intelligence applications for healthcare, with a certain degree of dependability. Uranus is capable of providing some self-organizing capabilities through the use of monitors such as a Connection Monitor and a Battery Monitor. Uranus has been used in the development of a Long-Term Monitoring application and a Smart Hospital application.

3. MidBlocks: A Component Supervising Middleware

As mentioned before, a solution to improve the reliability of a system is to perform supervision functions through the communication mechanism used for component interaction. Examples of this type of solution are the pneumatic control systems used in early building automation and the popular 4-20mA current-loop popular in industrial measurement and control systems.

In early building automation systems compressed air was used to transmit information between the components of the system and simultaneously supervise the system. To achieve this, components were linked with hoses that carried compressed air which varied in the range of 3-15psi. 3psi represents a live zero while 15psi represents 100%. Any pressure below 3psi is a dead zero and constitutes an alarm condition [8].

The same principle is used in the 4-20mA current-loop. In this scheme a current flowing through a wire to the sensor or actuator is varied between 4 and 20mA, with 4mA representing a live zero and 20mA representing 100%. If the current drops below 4mA this signals either a device failure or an open link while a current above 20mA signals a device failure or a short circuit [8, 9].

To supervise components in distributed systems, a frequent solution is to use heartbeat messages. In this type of scheme a special kind of message, called a heartbeat message, is periodically sent by the component to other components in the system. The main disadvantage of this type of scheme is that depending on the number of system components, the frequency with which heartbeat messages are sent and the bandwidth of the communications links, the load on the communications infrastructure can be significant.

The event based store and forward middleware is particularly well suited to provide a supervising mechanism without the need of a separate heartbeat message. Events generated by the system components can be considered as a type of heartbeat message. This coupled with the need of an event management entity for the store and forward mechanism, which can simultaneously be used to supervise the interval between events, provides the basis for a complete component supervision solution. This is the principle we use in MidBlocks, an event based, store and forward, component supervising middleware.

3.1. MidBlocks Description

MidBlocks is an event based, store and forward type of middleware in which the store and forward entity additionally performs component supervision tasks. Every component of a MidBlocks based system specifies a maximum time interval between the events that it generates. If a component exceeds this maximum time interval, called the Maximum Event Interval (MEI), MidBlocks will generate a failure event notifying the component's failure to those components that have subscribed to failure events.

A useful feature of MidBlocks is that it does not assume any particular type of communication mechanism or underlying hardware. To achieve this MidBlocks makes use of a layered design in which events are formed independent of any communications mechanism or hardware assumptions and as the events descend through the layers information required by the specific communication mechanism and hardware chosen is added. As a result MidBlocks can be used to provide component supervision in a large array of different systems.

In order to use MidBlocks in a particular system only two elemental requirements have to be satisfied:

- Each component of the system must be uniquely identifiable.
- A communication link must be present, capable of sending and receiving discrete data packets between the components and the store and forward/component supervising module.

3.2. MidBlocks Events

All components in MidBlocks must be assigned a unique identifier. This way event messages can be related to the component from which the message originated. Component identifiers in MidBlocks are 48 bits long analogous to the MAC address used in IEEE 802 networks. These identifiers can either be assigned statically during system installation or dynamically if an appropriate component registration mechanism is provided. The only requirement is that the identifiers are system wide unique.

All MidBlocks events have the same basic structure which is the 48 bit component id, followed by an 8 bit event type id and, depending on the type of event, additional event data (figure 1).

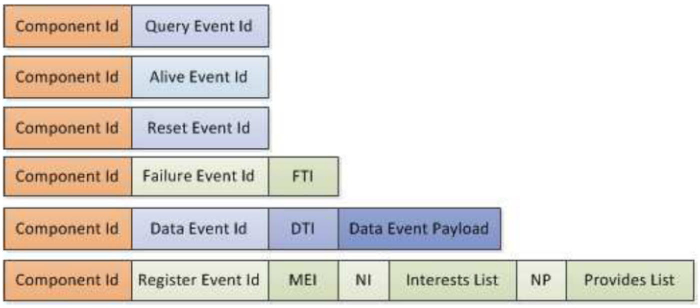

Figure 1. MidBlocks event structure.

MidBlocks uses the following six types of events:

- *Query events.* When a component has exceeded its maximum MEI the Component Supervisor (CS) module sends a query event to the component to determine if it is responsive. If the component does not respond within an MEI then a critical failure event is sent to those components subscribed to failure events.

- *Alive events.* There are two type of Alive events with different event type id that can be issued by components. The first one is issued in response to a query event. While the second one is issued when the component has not generated any events and its MEI is close to expiring.

- *Reset events.* When an event is received by a component that is not registered with the Message Receiver then a Reset event is issued immediately back to the component. Upon reception of a Reset event the component should clear any pending outbound events and perform its registration procedure.

- *Failure events.* Failure events are generated when messages are not received or cannot be delivered to components. Failure events have an 8 bit Failure Type Id (FTI) which indicates the type of failure detected.

- *Data events.* Data events are the normal events exchanged between components and the store and forward entity in MidBlocks. After the event type id, Data events have a 64 bit Data Type Id field. The possible values of the DTI are application specific and are assigned by the system developers using MidBlocks. After the DTI field comes the actual event data. MidBlocks does not assume any particular structure or size for this data.

- *Register events.* Register events are used to register components with the store and forward/component supervision entity. MidBlocks handles two values for the Register Event Id. The first value is used for components that have interest in receiving failure events, while the second value is used for components that do not need to be notified of failure events. Following the event type id Register events have the following fields:

    1. *MEI.* This is a 32 bit field containing the Maximum Event Interval for this particular component expressed in milliseconds. This value is sent using network byte order.

    2. *NI.* The Number of Interests field (NI) is a 64 bit field indicating the number of data event types that this component has interest in. This value is sent using network byte order.

    3. *Interest List field.* This field lists the DTIs of all the data types that this component is interested in receiving.

    4. *NP.* The Number of Provides field (NP) is a 64 bit field indicating the number of data event types that this component provides. This value is sent using network byte order.

    5. *Provides List field.* This field lists the DTIs of all the data types that this component provides.

3.3. MidBlocks Architecture

MidBlocks can be broadly separated into two distinct parts (see figure 2): A component side part (CMPS) and a Store and Forward/Component Supervising entity (SFCS). As with any other type of store and forward event-based middleware the primary task performed by MidBlocks is to deliver events from producers to consumers. However, unlike other types of middleware MidBlocks supervises the components to ensure that at least one event is sent during a components MEI and that events are being accepted by the components. To achieve this MidBlocks requires event message supervising functions which constantly monitor the events being accepted from components by the SFCP and the events delivered to components from the SFCP. In order to perform adequate supervision MidBlocks does not support direct component to component interaction.

In the following sections we will begin by describing the operation of the CMPS part of MidBlocks and afterwards describe the SFCS entity.

3.3.1. MidBlocks CMPS

When a component is first introduced into the system it must perform a registration procedure. The purpose of the registration procedure is to allow the SFCS to configure the queuing and component supervision mechanisms adequately to handle the events of this component and to supervise the component. Registration of the component with the SFCS is performed by the CMPS Component Registration Module (CMPS-CR) upon request of the component's main process (see figure 3). The registration sequence has to be initiated by the main process in order to ensure that the communications hardware and software have been properly initialized and are able to send and receive messages. Components must use one of two Register Event Ids to indicate whether or not they are interested in receiving Failure events.

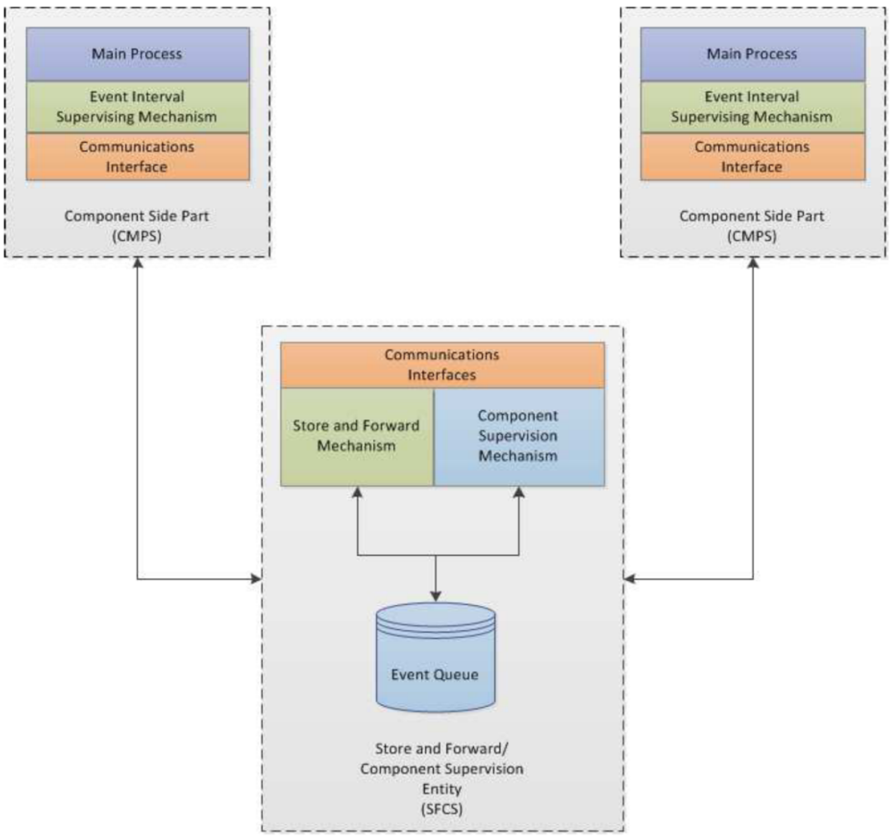
Main Process
Event Interval Supervising Mechanism
Communications Interface
Component Side Part (CMPS)
Main Process
Event Interval Supervising Mechanism
Communications Interface
Component Side Part (CMPS)
Communications Interfaces
Store and Forward Mechanism
Component Supervision Mechanism
Event Queue
Store and Forward/ Component Supervision Entity (SFCS)

Figure 2. MidBlocks architecture.

Inbound event messages arrive at the CMPS from the SFCS through a communications interface and are received by the CMPS Message Receiver (CMPS-MR). Four types of events can be received by a component: Query, Reset, Data, and Failure events. Failure events are received only by components that have expressed interest in receiving this type of events by using the corresponding Register Event Id during their registration procedure. Depending upon the type of event received by the CMPS-MR the following tasks are performed:

- *Query event.* When a Query event is received, the CMPS-MR immediately notifies the CMPS Message Dispatcher/MEI Supervisor (CMPS-MD) to send an appropriate Alive event. The Query event is then discarded and no further processing of the event occurs.
- *Reset event.* Upon reception of a Reset event the following sequence of actions occurs:
    1. CMPS-MR signals the CMPS-MD to block any further event message sending, with the exception of Register events.
    2. The CMPS-MR then introduces the event into the CMPS Local Event Queue (CMPS-EQ) and will discard any further Reset events until it is signaled by the CMPS-MD that a Register event has been sent.
    3. The CMPS-MD will not accept any events from the component's Main Process until it receives a Register event from the CMPS-CR.
    4. When the component's Main Process extracts the event from the queue it must signal the CMPS-CR to perform the registration procedure to register the component with the SFCS.
- *Data and Failure events.* Data and Failure events are placed directly in the CMPS-EQ for processing by the component's Main Process.

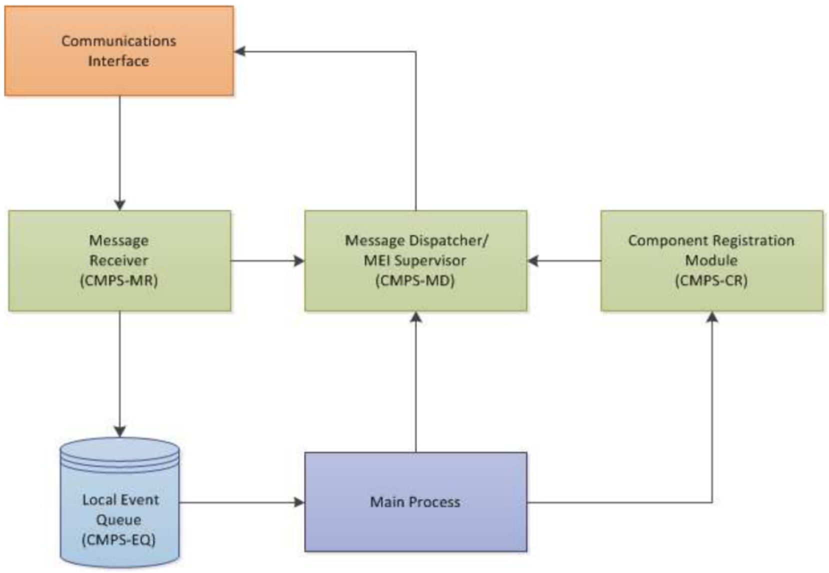
Communications Interface
Message Receiver (CMPS-MR)
Message Dispatcher/ MEI Supervisor (CMPS-MD)
Component Registration Module (CMPS-CR)
Local Event Queue (CMPS-EQ)
Main Process

Figure 3. MidBlocks component side (CMPS) block diagram.

Outbound data events are sent through the CMPS-MD and the communications interface. The CMPS-MD monitors the time interval since the last event was sent and, in case it approaches the component's MEI, it issues an Alive event to signal that the component is operating correctly.

3.3.2. MidBlocks SFCS Event Reception

On the SFCS side (see figure 4) event messages are received by the SFCS Message Receiver (SFCS-MR) from the communications interfaces. Three types of messages can be received by the SFCS: Alive, Register and Data events. The first task performed by the SFCS-MR is to verify the event type and the Component Id. If the event is not a Register event and the component's data is not registered in the SFCS Component Register (SFCS-CR) then a Reset event for the component is issued through the same communications interface from which the message arrived. Since Reset events are not entered into the SFCS Event Queue (SFCS-EQ) and are not sent by the SFCS Message Dispatcher (SFCS-MD) the data in the Component Id field of the Reset event is not important.

After verifying that the event is valid one of the following actions is performed depending on the event type:

- *Alive and Register events.* Alive and Register events are directly placed in the message queue without any further processing from the SFCS-MR.
- *Data event.* When the event received is a Data event the SFCS-MR scans the SFCS-CR to determine which components are subscribed to the type of data event received. Next the SFCS-MR attaches a Time-To-Live (TTL) field to the event, and inserts one copy of the event into the SFCS-EQ for each recipient. The inserted events are tagged as New events prior to insertion as explained in the following section.

3.3.3. MidBlocks SFCS Event Processing

Events in the SFCS-EQ can be tagged with four states:

- *New event.*
- *Ready for Delivery.*
- *Unable to Deliver.*
- *Processed.*

Any event in the SFCS-EQ can only be in one of these states at any given time.

The SFCS Component Supervisor (SFCS-CS) constantly scans the SFCS-EQ for events that require processing. Depending on the event type and the state of the event one of the following actions is performed:

- *New event tag - Alive event.* When the SFCS-CS encounters an Alive event tagged as a New event, it resets the timer associated with the producer component and tags the event as Processed. If the Alive event was issued in response to a Query event then the SFCS-CS inserts a Failure-Non critical event into the SFCS-EQ for each Failure event subscriber. Failure events have the same structure as Data events in the SFCS-EQ meaning that the event has a TTL field and a state tag. When the Failure event is created it is tagged as Ready for Delivery.
- *New event tag - Data event.* Data events are handled in much the same way as Alive events. The SFCS-CS resets the timer associated with the producer component and tags the event as Ready for Delivery.
- *New event tag - Register event.* On encountering a Register event tagged as a New event, the SFCS-CS creates and initializes a new timer for the component and inserts the component's data into the SFCS-CR. The Register event is then tagged as Processed.

- *Ready for Delivery*. Events tagged as Ready for Delivery are not handled by the SFCS-CS but are handled by the SFCS-MD. How the SFCS-MD handles Ready for Delivery events is described later on in this section.
- *Unable to Deliver*. This tag indicates that the SFCS-MD was unable to deliver the event after TTL attempts. When encountering a message tagged as Unable to Deliver the SFCS-CS performs the following actions:
  1. The destination component's data is removed from the SFCS-CR.
  2. Any events in the SFCS-EQ destined for the component, including the current event, are tagged as Processed.
  3. Failure events of type Critical are inserted into the SFCS-EQ indicating that the component has failed.
- *Processed*. Processed are handled by the SFCS Garbage Collector (SFCS-GC). The SFCS-GC scans the SFCS-EQ for events tagged as Processed and removes the events from the SFCS-EQ.

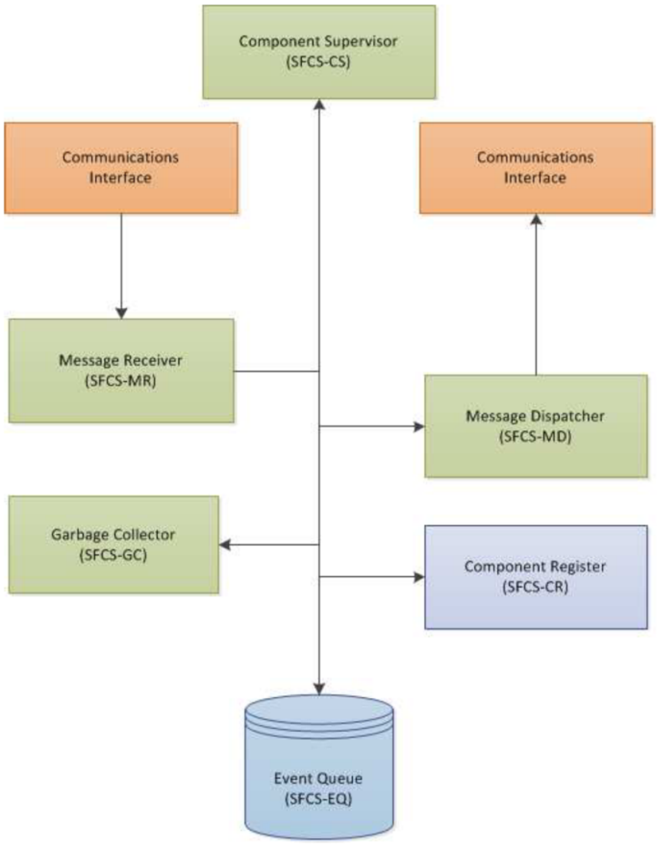
Component Supervisor (SFCS-CS)
Communications Interface
Communications Interface
Message Receiver (SFCS-MR)
Message Dispatcher (SFCS-MD)
Garbage Collector (SFCS-GC)
Component Register (SFCS-CR)
Event Queue (SFCS-EQ)

Figure 4. MidBlocks Store and Forward/Component Supervision (SFCS) entity block diagram.

The SFCS-CS also keeps track of the time elapsed since component's last event was received. If a component exceeds its MEI the SFCS-CS inserts a Query event destined for the component into the SFCS-EQ. Additionally the SFCS-CS inserts a non-critical failure event into SFCS-EQ and resets the timer associated with the component.

3.3.4. MidBlocks SFCS Event Delivery

Data events marked as Ready for Delivery are read from the SFCS-EQ by the SFCS-MD on a FIFO basis and delivered to consumers via the appropriate communications interface. If the event is delivered successfully then it is tagged as Processed, otherwise the event's TTL field is decremented by one. If the event's TTL field reaches zero then the event is tagged as Unable to Deliver and no further attempts will be made to deliver the event.

3.4. MidBlocks Communications Interfaces

To allow use with a wide variety of devices and communications technologies, MidBlocks employs a 3 layer design for its communication interfaces as shown in figure 5.

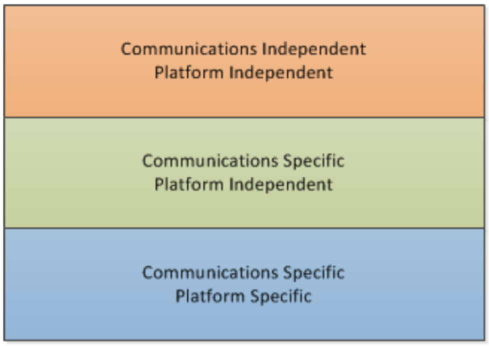

Figure 5. MidBlocks communications interface layers.

In the first layer event messages are formed without using communication specific data such as IP addresses or node ids. Once the core event messages are formed, communication specific parameters such as IP addresses are added to the message. Finally, in the last layer, platform specific routines are invoked to deliver the message.

Currently, the CMPS can only use a single communications interface in each component, however the SFCS can use multiple communications interfaces both for inbound as for outbound messages. This allows the SFCS to function as a gateway which permits components with different types of communication mechanisms to exchange events.

4. Current Projects with MidBlocks

A reduced version of MidBlocks is currently being used in FunBlocks [10-12]. FunBlocks is an event based, minimalist, modular framework for the development of Ambient Intelligence systems. Figure 6 shows a diagram of the basic components of FunBlocks

FunBlocks approaches the development of AmI systems from the point of view of distributed control systems. FunBlocks makes use of the function block abstraction described in the IEC 61499 standard for distributed control systems [13, 14].

Developing an AmI system using the FunBlocks framework consists of developing or reusing function blocks which provide the desired functionality, and afterwards joining these blocks through the services provided by the framework.

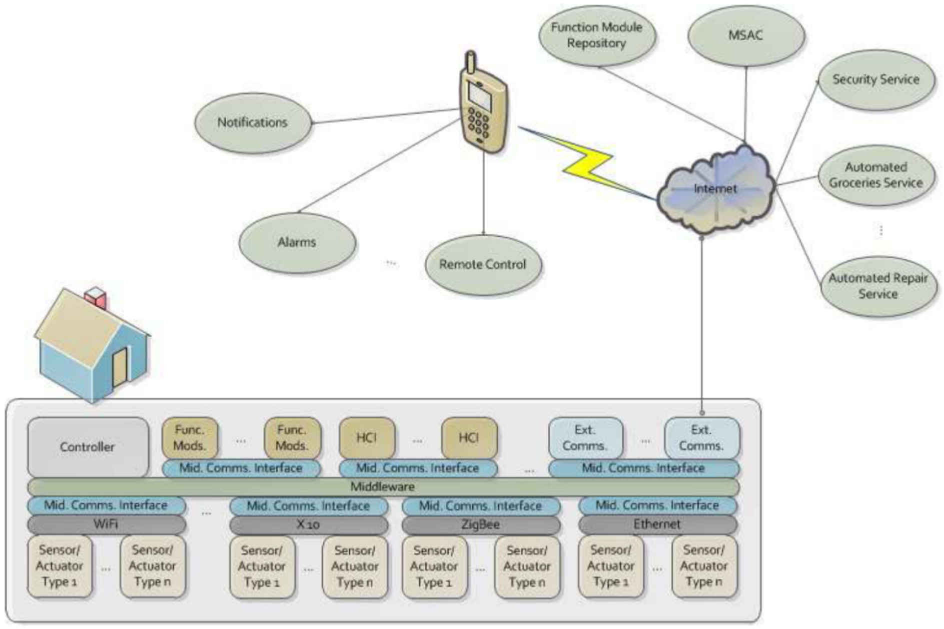

Figure 6. FunBlocks diagram

Thanks to its communications technology independent design, the use of MidBlocks allows FunBlocks to handle different types of sensor and actuator communications protocols such as WiFi, ZigBee and RS-485 buses.

5. Conclusions and Future Work

Although MidBlocks novel design increases the reliability of a system due to its components and communication link supervision functions a weak link in the design is the use of a single Store and Forward/Component Supervising entity. The SFCS represents a single point of failure in the system. To improve the reliability of a MidBlocks based systems the design of MidBlocks must be improved to accommodate redundant or cooperating SFCS.

Asides from the reliability increase to MidBlocks based IEs, the use of multiple SFCS brings added flexibility not currently available in MidBlocks. The use of multiple SFCS would allow, for example, the development of IEs with self-configuring and self-healing capabilities.

References

[1] L. Roalter, A. Möller, S. Diewald, and M. Kranz, Developing Intelligent Environments: A Development Tool Chain for Creation, Testing and Simulation of Smart and Intelligent Environments, *Seventh International Conference on Intelligent Environments*, Nottingham, United Kingdom, 2011, 214-221.

[2] A. Avižienis, J.-C. Laprie, and B. Randell, Fundamental Concepts of Dependability, Newcastle University CS-TR-739, 2001. Available: http://www.cs.ncl.ac.uk/research/trs/papers/739.pdf (May 5, 2012).

[3] P. A. Bernstein, Middleware. A Model for Distributed System Services, *Communications of the ACM* **39 (**1996), 86-98.

[4] S. Tarkoma and K. Raatikainen, State of the Art Review of Distributed Event Systems, Helsinki University Computer Science Department, 2006. Available: http://www.minema.di.fc.ul.pt/reports/MinemaEventsReport_final.pdf (May 5, 2012).

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, The Many Faces of Publish/Subscribe, *ACM Computing Surveys* **35** (2003), 114-131.

[6] A. C. Rice and A. R. Beresford, Dependability and Accountability for Context-Aware Middleware Systems, *Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops*, 2006, 378-382.

[7] A. Coronato, Uranus: A Middleware Architecture for Dependable AAL and Vital Signs Monitoring Applications, *Sensors* **12** (2012), 3145-3161.

[8] Building Automation Products Inc., Understanding 4-20 mA Current Loops [Application Note]. Available: http://www.bapihvac.com/CatalogPDFs/I_App_Notes/Understanding_Current_Loops.pdf (Jan 14, 2012).

[9] National Semiconductor, Application Note 300. Simple Circuit Detects Loss of 4-20 mA Signal [Application Note 300]. Available: http://www.ti.com/lit/an/snoa605a/snoa605a.pdf (Jan 14, 2012).

[10] R. Baquero, J. G. Rodríguez, S. Mendoza, and D. Decouchant, Towards a Uniform Sensor Handling Scheme for Ambient Intelligence Systems, *8th International Conference on Electrical Engineering, Computing Science and Automatic Control (CCE 2011)*, Mérida Yucatán, México, 2011, 952-957.

[11] R. Baquero, J. G. Rodríguez, S. Mendoza, and D. Decouchant, Towards a Modular Scheme for the Integration of Ambient Intelligence Systems, *5th International Symposium on Ubiquitous Computing and Ambient Intelligence (UCAmI 2011)*, Riviera Maya, Mexico, 2011.

[12] R. Baquero, J. G. Rodríguez, S. Mendoza, and D. Decouchant, FunBlocks. A Modular Framework for AmI System Development, *Sensors,* 2012 (Submitted).

[13] R. Lewis, *Modelling Control Systems Using IEC 61499*, The Institution of Engineering Technology, England, 2008.

[14] V. Viatkin, *IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design*, International Society of Automation, Research Triangle Park, North Carolina , U.S.A., 2007.