

# Architecting RFID Middleware

Joseph E. Hoag and Craig W. Thompson • University of Arkansas

**R**adio frequency identification (RFID; <http://en.wikipedia.org/wiki/RFID>) is an automatic identification technology that relies on cheap tags (transponders) that can be attached to objects (such as pallets, containers, ID badges, assets, vehicles, and pets) and that nearby readers can track. Although it isn't new, RFID technology has dramatically improved in recent years, resulting in lower-cost tags (now less than 10 cents) and higher read reliability.

Commercial RFID is particularly interesting because we can view it as a vanguard of the coming world of sensors in which most or all individual objects will have network identities and interfaces through which they'll communicate wirelessly with humans and other objects. Past Architectural Perspective columns have covered this coming "Internet of things" in which "everything is alive."<sup>1-3</sup>

Because RFID is in the fairly early adoption stage, many companies are still trying to assess the business case for it. A recent study<sup>4</sup> suggests that RFID can significantly help with the common retail-industry *out of stocks* problem (products that should be on the shelf but are not). Early adopters also face the problem of determining which hardware and software investments to make because every RFID reader and printer presents a different interface and systems aren't yet interoperable.

In summer 2005, the Computer Science and Computer Engineering Department at the University of Arkansas started a project aimed at developing open-source RFID middleware to address this latter problem. In this article, we describe that software's development and the challenges and benefits of large scale software development in universities.

## Overview of the RFID Middleware Architecture

In a typical warehouse, trucks back up to a dock door to unload pallets of goods. The pallets and con-

tainers generally have RFID tags but, at least today, few low-cost items have their own tags (for cost and consumer-privacy reasons). RFID readers might be situated throughout the warehouse at places such as dock doors or doors leading into stores. By reading tags, organizations can trace their pallets and containers as they move throughout the supply chain.

A first challenge we faced in developing our middleware was in how to model the warehouse world. We decided to use a mixture of agent and object technologies, making each device (RFID reader, printer, camera, motion detector, forklift, conveyor belt, sensor, and so on) an agent. Although device interfaces vary widely in terms of formats and protocols, all our agent communications occur via XML messages over UDP/IP. Agents act as "wrappers" or "translators" for their associated devices: other middleware communicates with the agent, and the agent communicates with the device. Figure 1 shows several reader agents and a printer agent communicating with a database management system (DBMS) agent and some user interface agents.

Soon after starting our development, we discovered that we needed to develop two software layers: a general-purpose core architecture, extensible for pervasive computing applications, and an RFID-related application architecture that specializes the generic agent architecture. We code-named the agent architecture "Ubiquity" and the RFID application architecture "TagCentric."

## Ubiquity Agent Architecture

We had several goals in mind when we created the Ubiquity agent architecture: flexibility, portability, fault-tolerance, and power. Let's take a look at the architecture to see how those are accomplished.

## Core Architecture Classes

As illustrated in Figure 2, the agent is the funda-

mental building block in the Ubiquity architecture. (In the XML diagrams in this article, bolded-italicized items are abstract methods.) The agent class provides basic communication functionality: subscribe to a message type, publish a message, post a message to a specific destination (which does not result in a return message), send a message to a specific destination (which results in a return reply), and handle an incoming message. The only things that distinguish one agent from another are which message types the agent handles and how it processes them. You create an application-specific agent by creating an agent-derived class that properly handles a given set of messages.

The transport class handles all communications for an agent. We use two basic flavors of messages: *point-to-point* messages involve a single recipient (for example, a command to turn on a reader), and *publishable* messages are “general interest” messages that could have any number of interested recipients. For example, an agent sends an `AgentConnected` message when it starts up. Rather than implement a complicated publish-subscribe mechanism for publishable messages, we decided to use a multicast facility. A transport object maintains two sockets: a Multicast socket handles publish-subscribe operations for publishable messages, and a regular UDP socket handles post-send operations for point-to-point messages.

The transport class employs background threads to “unload” incoming packets from its sockets and store them on intermediate queues. By dedicating threads to the quick unloading of incoming packets, we avoid packet-overrun problems on the receive side. We also built fragmentation logic into the Transport class, so that messages of any size can be successfully transmitted and received.

The XMLTranslator class assists the transport class in translating incoming XML messages into message objects.

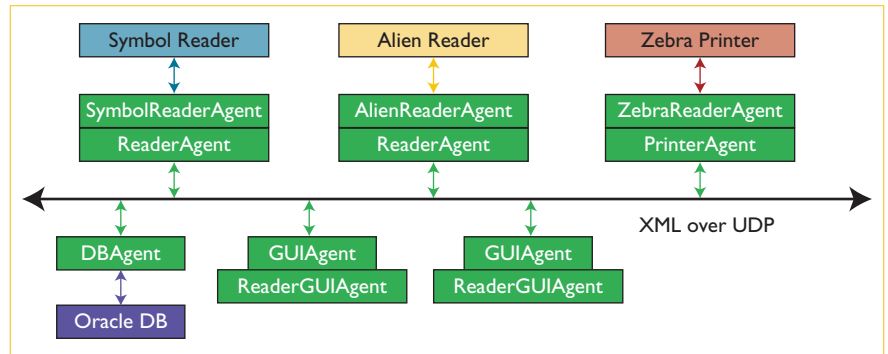


Figure 1. Sample TagCentric configuration. Logical devices are connected by XML-based message passing. Adding new device types is easy in this architecture.

### Messaging

Figure 2 shows our messaging scheme, along with a few of our architecture-level messages. The basic idea is that each message contains “header” information, which corresponds to the message class, and a type-specific message body, corresponding to a MessageBody-derived class.

Each message class (both header and body) contains a `toXML()` method, which converts a message object to an XML string. We use the XMLTranslator class to convert from XML strings to message objects. To simplify the XML-to-object transformation, we design message classes so that each MessageBody-derived class must fill in a value for the `MessageType` field in its constructor, and for each attribute `XYZ` of the message class, a corresponding `setXYZ()` method must accept a string parameter representing the value of `XYZ` and generate a return type of `void`.

A complementary processes converts a message object to XML and converting XML to a message object (in this case, a `DiscoveryRequest` message). The sending agent for a message will use the message object’s `toXML()` method to convert it to an XML string. The receiving agent will employ the `XMLTranslator.parse()` method to convert the incoming XML string to a message object.

### User Interface Support

Our generic device architecture takes

advantage of the soft controller described in an earlier column.<sup>3</sup> A soft controller is a truly universal remote control that can read a remote device’s user interface specification. There might be more than one such specification for a given device, based on the model-view-controller design pattern. That is, devices support native API interfaces but provide separate human interfaces, called GUIpanels. As Figure 2 illustrates, the GUIpanels for different devices are arranged in a GUIagent dashboard such that each GUIpanel corresponds to a connected agent.

The GUIagent operates by listening for `AgentConnected` messages. When it receives one, it searches for the existence of a `<TYPE>Panel` class, where `<TYPE>` is the value of the `Type` field in the `AgentConnected` message. If it locates a `<TYPE>Panel` class, the GUIagent creates a new panel corresponding to the newly connected agent. If a `<TYPE>Panel` class doesn’t exist, the agent ignores the `AgentConnected` message. For example, if an agent of type `Reader` emitted an `AgentConnected` message, the GUIagent would look for a `ReaderPanel` class and pop one up if it existed.

Figure 3 provides a UML description of the GUI classes in the architecture. GUIagent is an abstract class; an application-specific class derived from it must provide the `setUpLocalPanels()` method. A local panel, such as

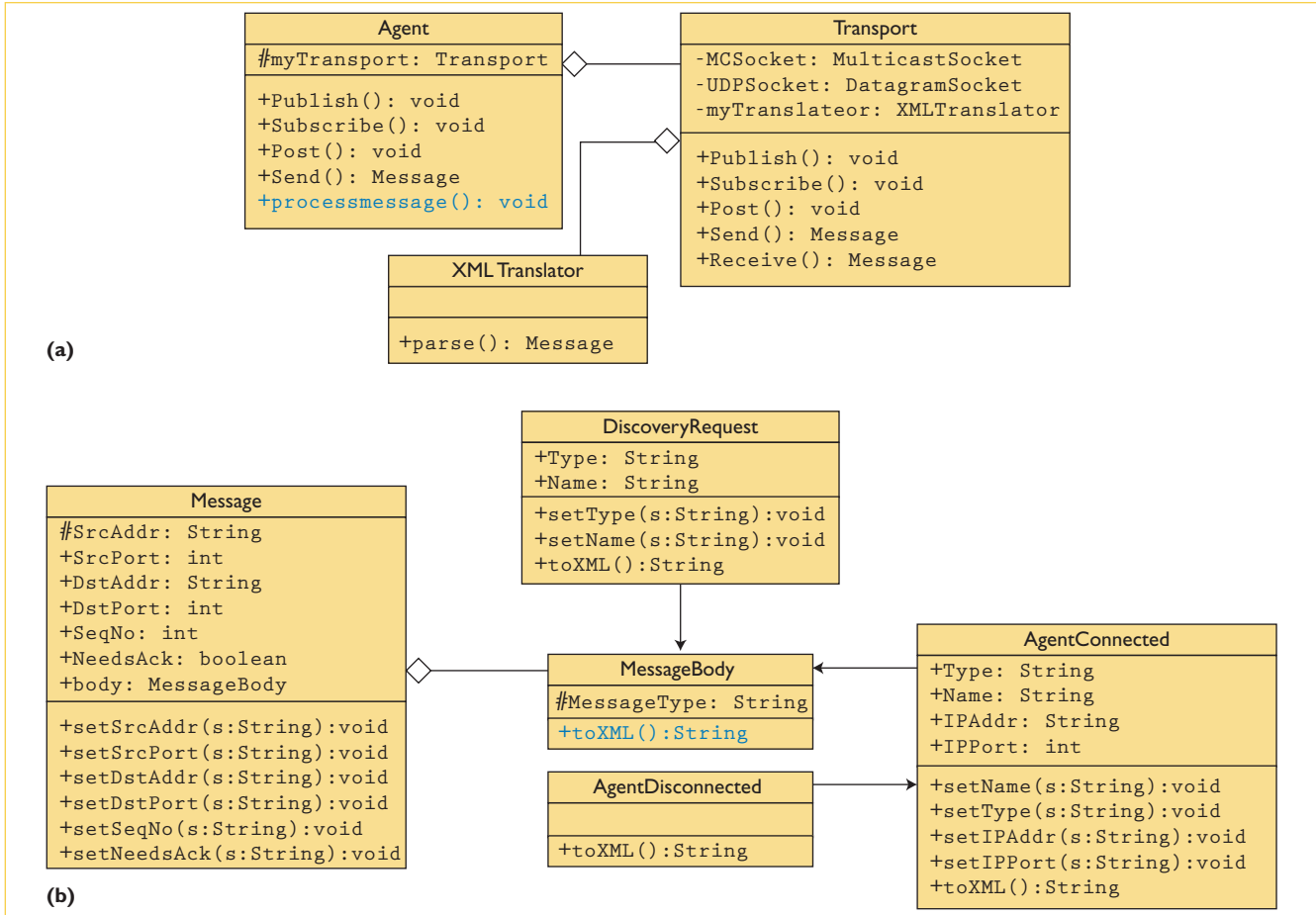


Figure 2. Classes in the architecture. At a most abstract level, the architecture is based on agents and messages sent between agents.

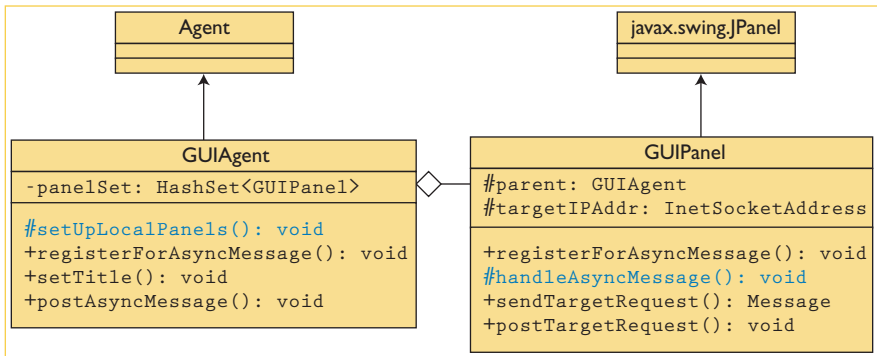


Figure 2. Architectural GUI support. At the architectural level, an agent can be associated with potentially several GUI panels that can be used to control the agent. (formerly Fig. 3)

of incoming asynchronous messages. GUIpanel-derived classes can register for asynchronous (i.e., publishable) messages via the registerForAsyncMessage() method. GUIpanel-derived classes let application developers associate panels with agent types; a panel for agent type XYZ is displayed for each XYZ agent that becomes active. Figure 3 shows a ReaderPanel from our TagCentric application. Through GUIpanels and agent typing, you can develop a common look and feel for entire classes of devices, such as RFID readers or RFID tag printers.

## TagCentric RFID Application Architecture

Our initial goal was to be able to gather tag-read information from RFID

an administration or test panel, doesn't correspond to any specific agent; local panels are typically application-specific. The application can also set the GUIagent window title.

The GUIpanel class is also abstract; the application-specific GUIpanel-derived classes must provide a handleAsyncMessage() method to determine the disposition

readers and deposit the information into a user-specified database. This meant that we needed to be able to communicate with several different reader types and databases. Belatedly, we also realized that we needed to support RFID tag printers because you can't read tags unless you print them first. At present we support

- three popular RFID tag-reader types (Symbol, Thingmagic, and Alien), all of which present different native APIs;
- several relational DBMSs (Derby, Oracle, DB2, MySQL, Sequel Server, and Postgres), using a Java database connectivity (JDBC)-based standardized interface; and
- one popular RFID tag printer (Zebra).

### Agents

Figure 4 shows a UML representation of the agent-derived classes in TagCentric. The abstract ReaderAgent class provides some common logic used across all readers and (through the use of abstract methods) dictates that all ReaderAgent-derived classes must properly respond to ReaderOn, ReaderOff, and ReaderStatusRequest messages. In addition to providing agents for Symbol, Alien, and Thingmagic readers, we provide a synthetic reader agent (FakeReaderAgent), which is useful for testing when you can't access a real reader.

Similarly, the abstract PrinterAgent class provides some common logic across all printer types, and also dictates that all PrinterAgent-derived classes shall support the PrinterStatusRequest and PrintTags messages.

We chose to handle the database agent a little differently. Because so much of what a DBagent does is the same across all database types, we implemented a single non-abstract DBagent class, rather than deriving a class for each specific database (such as OracleDBAgent, DerbyDBAgent). The DBagent must respond to DBSta-

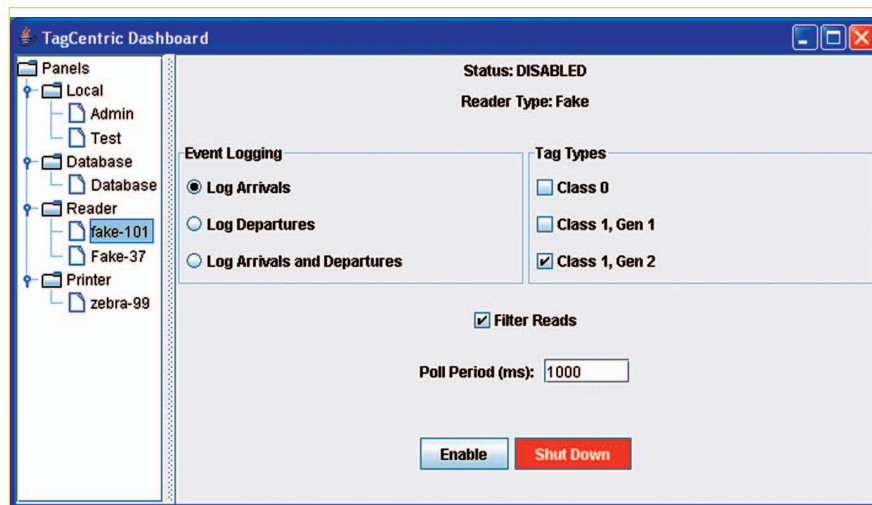


Figure 3. GUIagent from TagCentric application. On the left is a tree-based list of currently active agents, organized by type. On the right is the GUIpanel for the currently selected agent. (formerly fig 4)

tusRequest, DBinitialize, DBquery, DBCmd, and TagDataList messages.

### User Interface

Figure 5 shows the GUI-related classes associated with TagCentric. The RFIDGUIagent is the GUIagent-derived user interface application (see Figure 4).

The AdminPanel lets users manage reader and printer agent information, as well as launch (or kill) the database agent or any number of reader or printer agents. The TestPanel lets users conduct RFID-related performance tests.

The system displays the ReaderPanel, DatabasePanel, and PrinterPanel on demand, whenever a ReaderAgent, DBAgent, or PrinterAgent (respectively) is activated. In Figure 5, two reader agents, one printer agent, and one database agent are active, for example, and their panels are available for user interaction.

Note that there's only one ReaderPanel, rather than one for each type of reader. This means that TagCentric presents the same user interface for all RFID readers, which greatly simplifies the use of the software for those facilities that support multiple reader types. The same general scheme will apply to tag printers when we support multiple tag printers.

As we go to press, TagCentric has been operational for a semester at the University of Arkansas RFID Research Center. We have lately added user-defined features such as an easy-to-use install script, query histories, DBMS schema evolution, and documentation, and we've tested scalability with 10 to 20 readers of mixed types. Our TagCentric software should be ready for open-source release in the Fall 2006 semester.

We're planning several enhancements to the Ubiquity framework, including aspects of scalability, security, and reflection, as well as the TagCentric RFID application architecture, including intelligence, interoperability, and potentially an EPC Information Services reference implementation. We invite community contributions.

Our current challenge is code development in the face of high turnover — the average student spends one to two semesters on the project — so we have to assign modular units of work. Having one lead designer on the project from the beginning has thus been invaluable. In evolving from simply supporting the RFID Center to making our code open source in the Fall 2006 semester, we've engaged in a flurry of activity on documenting and testing. To

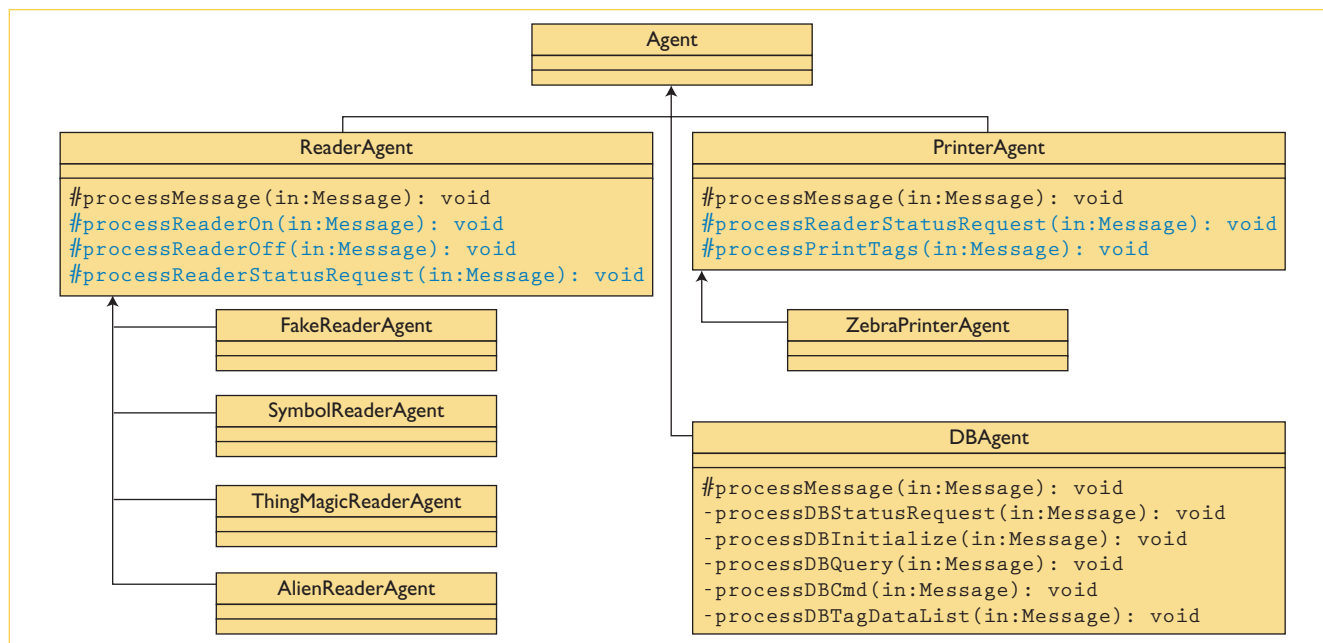


Figure 4. Abstract TagCentric classes for readers, printers, and DBMS systems are specialized by particular kinds of these devices. (formerly fig 5)

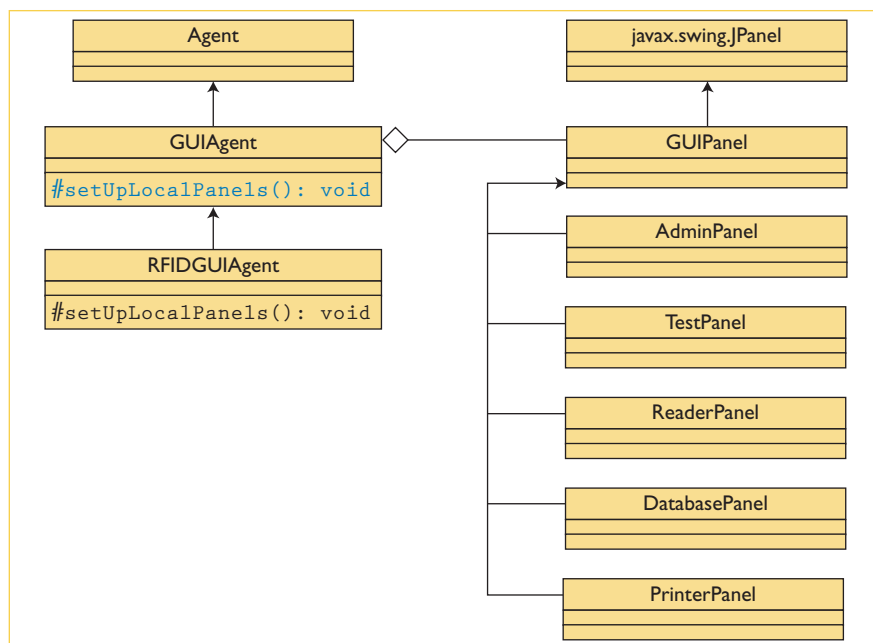


Figure 5. TagCentric user interface. We use a dashboard model for TagCentric user interface classes where each device family adds a panel. (formerly fig6)

ease into this transition to open source, we'll partner with some early adopters. We're unsure how this transition will affect our resources, so stay tuned and we'll have an update soon. ☐

References

1. C. Thompson, "Everything is Alive," *IEEE Internet Computing*, vol. 8, no. 1, 2004, pp. 83-86.
2. C. Thompson, "Smart Devices and Soft Controllers," *IEEE Internet Computing*, vol. 9,

no. 1, 2005, pp. 82-85.

3. C. Thompson, P. Pazandak, and H. Tennant, "Talk to your Semantic Web," *IEEE Internet Computing*, vol. 9, no. 6, 2005, pp. 75-78.
4. W. Hardgrave, M. Waller, and R. Miller, *Does RFID Reduce Out of Stocks? A Preliminary Analysis*, tech. report ITRI-WP058-1105, Information Technology Research Center, Nov. 2006; <http://itri.uark.edu/research/display.asp?article=ITRI-WP058-1105>

**Craig W. Thompson** is a professor and Charles Morgan Chair in Database at the University of Arkansas and president of Object Services and Consulting, a middleware research company. His research interests include data engineering, software architectures, middleware, and agent technology. Thompson has a PhD in computer science from the University of Texas at Austin. He is an IEEE Fellow. Contact him at [cwt@engr.uark.edu](mailto:cwt@engr.uark.edu).

**Joseph E. Hoag** is a PhD student at University of Arkansas. He has worked at Lawrence Livermore National Laboratory, Philips Electronics, Orbital Sciences Corporation, Boeing, and as a contract software engineer. Hoag has an MS in computer science from the University of California at Davis. Contact him at [jhoag@engr.uark.edu](mailto:jhoag@engr.uark.edu).