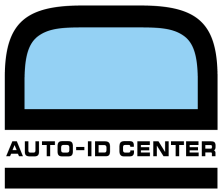
Published April 2003.



TECHNICAL REPORT

Savant™ Guide

Amit Goyal

AUTO-ID CENTER MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 400 TECHNOLOGY SQUARE, SIXTH FLOOR, CAMBRIDGE, MA 02139-4307, USA

ABSTRACT

This document describes the Savant™ for the purpose of making it more useable. Not only does it discuss parts of the Savant™ architecture, it also refers to the specific Savant™ v0.1 installation and highlights important configuration files. For experienced technical users, the document also provides source code examples to ease the application development process.

Published April 2003.

TECHNICAL REPORT

Savant™ Guide

Biography

Amit Goyal
Master's Candidate, MIT Lab

Amit Goyal received his Bachelor's in Computer Science from the Massachusetts Institute of Technology. Afterwards, he was a founding member of Isovia and a lead developer on their technology team. Most recently, Mr. Goyal completed an extended internship at Sun Microsystems Laboratories where he prototyped Sun's first Auto-ID offering, the Savant Appliance. His research at the lab focuses on addressing the necessary issues within the Auto-ID technical infrastructure to automate the business process known as shipping and receiving verification.

TECHNICAL REPORT

Savant™ Guide

Contents

1. INTRODUCTION

The current Savant™ Installation is meant for experienced technical users only. The download wizard simplifies Savant™ installation, but there are a number of factors which complicate using the Savant™. For example, the Savant™ software does not come with any frills such as GUIs, applications, or other software to assist the user. Additionally, a barebones system would require a reader and tags. Because users are only downloading the Savant™ and not purchasing the evaluation kit, it is likely this hardware will be missing.

2. BACKGROUND

2.1. What is the Savant™?

If each object currently using a barcode were to make use of the Auto-ID Center EPC™, then the managing infrastructure would have to handle millions of events every second. To manage this massive flow of events, the Auto-ID Center has proposed the use of modular components, stacked in a hierarchy, called Savants™. In such a system, the lower level Savants™ will process, filter, and digest events. To reduce network traffic, a Savant™ may just forward events of interest or event summaries to higher level Savants™.

The Savant™ has 3 main components: Real-time in-memory event database (RIED), Task Management System (TMS), and Event Management System (EMS). RIED is an optimized database that achieves some of its performance gains by supporting a limited subset of SQL. TMS coordinates processes initiated by higher level Savants™. EMS connects readers to applications by managing the event flow generated by the reader. When discussing the Savant™, the EMS is most often the subject. It is the most tangible part of the system, and it provides a platform on which users may compose interesting applications. This document will focus on the EMS and describe how users can develop applications for it.

2.2. Event Management System

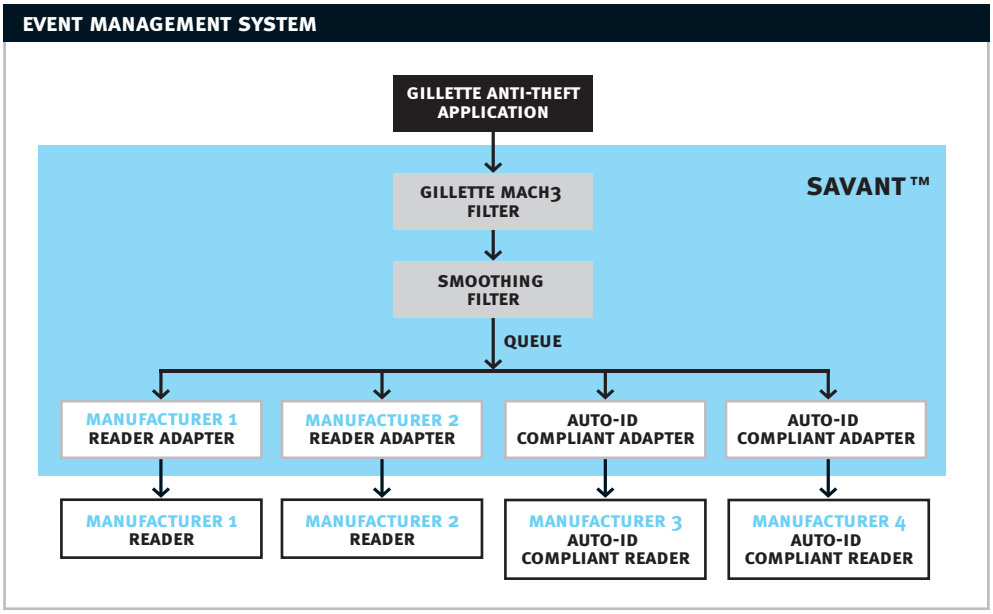


Figure 1: The Savant's™ Event Management System abstracts the implementation details from applications through a plug-in reader adapter architecture.

The above diagram illustrates how events are propagated through the Savant™ system. Everything above the light blue portion labeled "Savant™" comprises the entity we refer to as the Savant™ Event Management System. As illustrated at the base of the diagram, many readers are capable of connecting to the Savant™. A reader is a device that generates events and only communicates existence information. For example, it will state that a tag is present, but it will not state that the tag is absent. Absence is a higher-level notion that the Savant™ will have to determine on its own. Even if the tags in the reader's field do not change, the reader continues to emit a steady event stream indicating all the tags it sees. In addition to announcing EPCs™ and the associated timestamp at which those EPCs™ were seen, a reader has the ability to emit non-EPC™ information (temperature, humidity, etc.) as well as status events (tells the Savant™ system a single pass of the reader's field has just completed).

These readers may connect to the computer world through a number of different physical means (serial, Ethernet, etc.), and they may speak different, proprietary languages. To support readers of various types, the Savant™ uses manufacturer and reader specific reader adapters that handle the implementation details of connecting to different readers. If the reader is Auto-ID compliant, then only a single Auto-ID compliant adapter, independent of the manufacturer, is necessary.

After events are produced by the readers and consumed by the reader adapters, they are forwarded to a queue. In this queue, events are automatically forwarded to filters. Depending on how the filter is defined, certain events may be "filtered" out.

Filters

Filters separate out events and forward the relevant ones to other filters, loggers, or applications. Filters may be connected in Lego-like fashion to produce a desired cumulative behavior. For example, a smoothing filter in combination with a product filter might be used to isolate the events appropriate for an anti-theft application, in much like the manner illustrated in the Event Flow Model diagram below.

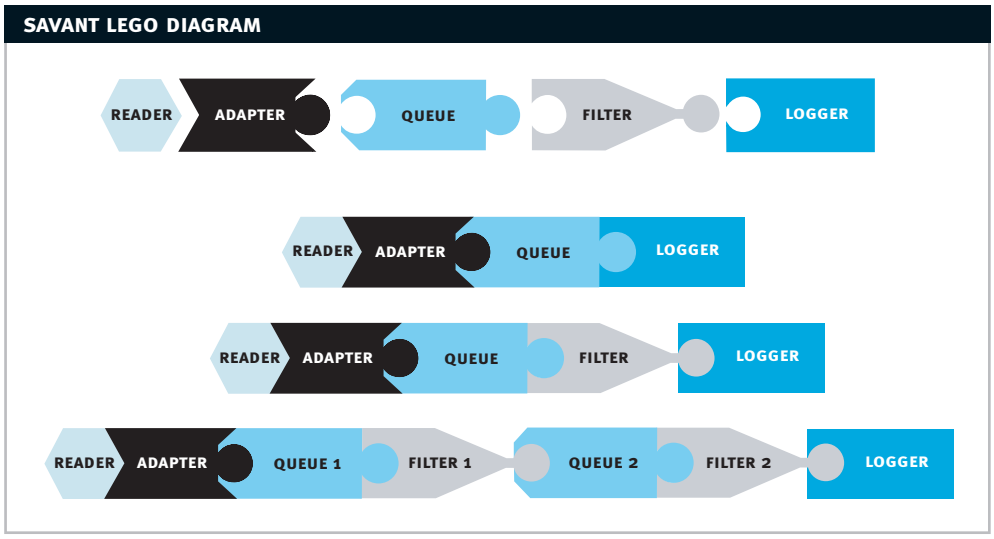


Figure 2: Filters are lego like components that can be reused to produce a cumulative behavior. Diagram by Mark Harrison.

Many different types of filters can be useful. Three obvious examples are the Product filter, the Temporal filter, and the Smoothing filter. Product filters may forward only the events associated with a particular item or manufacturer. Technically, this means the filter forwards events with EPCs™ that only fit within a specific range or pattern. Temporal Filters would filter events based on their timestamps. One temporal filter, for example, might only communicate the events seen in the last 10 minutes.

Smoothing filters exist because on any given pass, a reader may miss a few tags. As a result, raw events must be processed before a decision is made as to whether a tag truly has left the field. Such a decision is based on specifying a threshold number of passes that the reader fails to see a specific tag. If our threshold is 5, then we consider a tag missing if for 5 consecutive passes, a reader does not observe the tag. Below, in "Appendices 1 and 2 show examples of filters I have written."

Loggers
Loggers are essentially the same as filters, except they don't forward events to other filters or loggers. Loggers are generally used to store events in a database or to send events over some sort of network connection (socket, http, etc.). Network loggers are one manner through which the Savant™ communicates to applications. Other loggers of interest could generate diagnostic information to a system console or a file.

Applications
Applications rest on top of the Savant™ and are abstracted away from the implementation details of readers. The Savant™, through its system of filters, chooses the events that an application should see. To actually send events to the application, the programmer is free to do so in any manner he chooses. The Savant™ technical manual does not enforce the communication method. One might, for example, define a network logger that sends events over a socket connection to the receiving application. In such an application, an asynchronous intermediary buffer might also be used to ensure all the events are communicated to the application.

Readers
Adding readers to the Savant™ is a two step process. First, the reader specific adapter must be written, compiled, and referenced in the classpath variable used to start the Savant™. Then, the EMS configuration file must be amended to reference the reader and specify the queue onto which the events will be posted.

Reader Adapter
The reader adapter contains the specific implementation details needed for polling the reader and obtaining the events. The adapter must then categorize these events into either EPC™ events, non-EPC™ events, or status events, so they can be properly propagated through the Savant™ system.

Savant™-to-Savant™ Communication
Currently, there is not any defined way for Savants™ to communicate with each other. A Soap interface exists, but it is merely present to coordinate Savant™-to-Savant™ communication. For example, through the Soap interface users may add listeners. These listeners, however, are user defined. The Savant™ communication protocol, such as whether communication is over TCP/IP or SSL, and the contents of that communication, are all specified in listeners and user defined. Listeners can be filters and loggers, both of which have been described earlier.

3. USING THE SAVANT™

3.1. Developing Savant™ Applications

Savant™ applications may either be written in filters and loggers or they may sit outside the Savant™ framework. If doing the former, refer to section "Developing Filters and Loggers." If developing applications outside the Savant™ framework, a connection needs to be established between the Savant™ and the application. This connection may be developed on the Savant™ side in filters and loggers. For example, a filter could be

written that makes a TCP/IP connection with an anti-theft application. The filter would forward all the relevant events to the anti-theft application, and the application would then determine if a theft were occurring.

3.2. Developing Filters and Loggers

As noted in the background section, the only difference between filters and loggers is that loggers do not forward any events to other components filters or loggers. Filters and loggers must implement the ReaderInterface (refer to "The Savant™ Technical Manual," section 2.3.1). Refer to Appendix 1 for an example of a simple filter which forwards all its events.

Please note that from the code excerpt in Appendix 1, one cannot determine the module to which these events are being forwarded. The EMS.conf file, described below, specifies the structure of the lego-like system of filters and loggers.

If users want to write a filter that only looks for a specific product, then they could insert a condition limiting the events being forwarded to a specific EPC™ range. Refer to Appendix 2 for an example of such a filter.

3.3. Developing Reader Adapters

Reader adapters must implement the ReaderAdapterInterface (refer to "The Savant™ Technical Manual," section 2.3.4). Reader adapters which connect to physical readers will have to take into account the details of proprietary systems.

3.4. Adding Reader Adapters

Assuming the reader adapter is in a jar file, to add a reader adapter to the system and have the Savant™ recognize it simply requires the user to refer to the file and its location directly in the CLASSPATH variable defined in the Savant™ configuration file, mentioned below.

3.5. Adding Savant™ Readers

First install the reader adapter associated with this reader. If the reader adapter is already installed, then making the system recognize readers is relatively straightforward. The EMS.conf file must be modified to direct the events pulled from the reader and send them into their appropriate queues. Look at the Auto-ID Center's Savant™ Technical Manual Section 2.3.3 for exact details on the grammar of the EMS.conf file.

3.6. Important Configuration Files

There are a number of Savant™ configuration files worth mentioning. Most configuration files are located at: /usr/local/savant-0.1/conf. In addition to the configuration files, it is important to note that Savant™ logs are written to the following directory: /usr/local/savant-0.1/log. These logs are generated as a result of the Savant™ file described below.

Savant™
This is an important file because it defines the CLASSPATH variable with which the Savant™ runs. If trying to start EMS with a newly written filter, for example, it is important that the CLASSPATH in this file refers

to the directory in which that filter's class file resides. Jar files for the Savant™ typically reside in: /usr/local/savant-0.1/lib. If adding another jar file, it should be referenced directly by the CLASSPATH defined in /usr/local/savant-0.1/conf/savant so that it is picked up when the Savant™ is started.

This file also specifies the location where the log files are generated. System output is redirected manually here to one of the log files using the "›› " operator and directed to directory: /usr/local/savant-0.1/log.

In addition to being located in the conf directory above, a symbolic link for this file exists at /etc/init.d/savant. Type "service savant start", "/etc/init.d/savant start" or "/usr/local/savant-0.1/conf/savant start" to start the Savant™. Other arguments you may specify besides "start" are "stop" and "restart."

EMS.conf
This is the configuration file for the Event Management System. Upon startup, the Savant™ loads this file to instantiate the proper classes and determine the path in which events will propagate the system. It is here that one specifies the readers, reader adapters, filters, loggers, and queues to design a functioning Savant™. This Savant™ cannot support dynamically added filters and loggers. As a result, each time a change to the Event Management System needs to be made, the Savant™ must be stopped, this file modified, and then the Savant™ started again. To understand the grammar of the EMS.conf directory, refer to the Savant™ Technical Manual, available on the Auto-ID Center's website.

dcom.properties
This file specifies the properties for connecting to serial devices.

readers.properties
This file defines the properties for reader adapters used with the Savant™.

savant.properties
Created and defined at startup, I never actually needed to modify this file. It is, however, important should one decide to modify the connection properties of the PostgreSQL database. The password or user login name are examples of properties of the PostgreSQL database which one may want to change. After those changes are made to the PostgreSQL database, it is in this file that you specify the properties so the Savant™ connects to the database properly.

APPENDIX 1

ForwardAllEventsFilter.java

```
import org.autoidcenter.ems.*;
import org.autoidcenter.exception.EPMFException;

/*
 *   This filter merely forwards all the events it receives.
 *   It exists here as a base filter to illustrate how
 *   filters may be written.
 *
 *   @author: Amit Goyal
 */

public class ForwardAllEventsFilter implements EventFilterInterface {

   private ReaderInterface loggers[];
   public ForwardAllEventsFilter( String args ) {
       loggers = null;
   }

   public void logEpcEvent( long timestamp, String EPC, String
readerEPC ) {
       if ( loggers != null ) {
           for ( int i = 0; i < loggers.length; i++ ) {
               ReaderInterface logger = loggers[ i ];
               logger.logEpcEvent( timestamp, EPC, readerEPC );
           }
       }
   }
   public void logNonEpcEvent( long timestamp, String readingType,
String value, String readerEPC ) {
       if ( loggers != null ) {
           for ( int i = 0; i < loggers.length; i++ ) {
               ReaderInterface logger = loggers[ i ];
               logger.logNonEpcEvent( timestamp, readingType, value,
readerEPC );
           }
       }
   }

   public void logStatusEvent( long timestamp, String statusMessage ) {
       if ( loggers != null ) {
           for ( int i = 0; i < loggers.length; i++ ) {
               ReaderInterface logger = loggers[ i ];
               logger.logStatusEvent( timestamp, statusMessage );
           }
       }
   }
    public void setListeners( ReaderInterface loggers[] ) throws
EPMFException {
       this.loggers = loggers;
   }

   public void shutdown(){}
}
```

APPENDIX 2

RangeFilter.java

```
import org.autoidcenter.ems.*;
import org.autoidcenter.exception.EPMFException;

/*
 *    This filter forwards all the non-epc and status events it
 *    receives.  When receiving epc-events, it forwards only those
 *    events that start with 00000000B0000030000.  This EPC beginning
 *    may represent a particular product or range of products.
 *
 *    @author: Amit Goyal
 */

public class RangeFilter implements EventFilterInterface {

    private ReaderInterface loggers[];
    public RangeFilter( String args ) {
        loggers = null;
    }

    public void logEpcEvent( long timestamp, String EPC, String readerEPC )
{
        if ( EPC.startsWith( "00000000B0000030000" )) {
            if ( loggers != null ) {
                for ( int i = 0; i < loggers.length; i++ ) {
                    ReaderInterface logger = loggers[i];
                    logger.logEpcEvent( timestamp, EPC, readerEPC );
                }
            }
        }
    }

    public void logNonEpcEvent( long timestamp, String readingType,
String value, String readerEPC ) {
          if ( loggers != null ) {
              for ( int i = 0; i < loggers.length; i++ ) {
                 ReaderInterface logger = loggers[i];
                 logger.logNonEpcEvent( timestamp, readingType, value,
readerEPC );
            }
        }
    }

    public void logStatusEvent( long timestamp, String statusMessage ) {
         if ( loggers != null ) {
             for ( int i = 0; i < loggers.length; i++ ) {
                ReaderInterface logger = loggers[i];
                logger.logStatusEvent( timestamp, statusMessage );
            }
        }
    }

    public void setListeners( ReaderInterface loggers[] ) throws
EPMFException {
        this.loggers = loggers;
     }
     public void shutdown(){}
}
```

Designed and produced by Foxner. www.foxner.com