Securing Open Semantic Information Brokers in Smart Environment Systems
Implementation guidelines

Azza Kamal Nabih Hanafi
Senior R&D Engineer-SECC
azkamal@itida.gov.eg

Jarkko Kuusijärvi
Research Scientist-VTT Finland
jarkko.kuusijarvi@vtt.fi
Abstract

Smart environments refer to any type of environment (e.g., homes, offices, hospitals, schools) that is equipped with a variety of devices, sensors, and actuators that interact with inhabitants to provide them convenience and comfort. There is a variety of data sources in such environments that enable the development of rich application scenarios. Security is a key issue in such applications; systems should guarantee preserving users’ privacy and it should not allow any unauthorized entities to get access to any user or system information.

One of the promising technologies used in smart environments is semantic technologies. It is a way to enable data to be machine understandable. This way, data can be directly and indirectly processed by machines. To facilitate the development of such semantic smart environment applications, Smart-M3 architecture can be used. It is an open source architecture that consists of two main components: semantic information broker (SIB) and set of knowledge processors (KPs).

The SIB is the main repository for all information. It is important to secure SIBs and secure all its outgoing and ingoing communications. Not all SIB implementations have built-in security features as the technology is relatively new and is still under continuous development. It is important to have workarounds to raise the level of security while using un-secured SIBs. These workarounds are still valid if the SIB is secure, at least when integrating the SIB with some other services, such as web front end.

This technical report discusses some of these workarounds in a Smart-M3 based smart environment applications that 1) uses REST services to implement functionality on mobile based clients and 2) directly uses KP operations to implement the web application.

**Keywords:** Semantic Information Broker, SIB, Knowledge Processor, KP, Security, Certificate, Encryption, Authentication, Authorization.
# Table of Contents

1. Introduction ........................................... 4
2. Smart-M3 Architecture ............................ 6
3. System Development Environment ............ 7
4. Solution Strategies .................................. 7
5. SIB Access Restrictions – Set Firewall Restrictions ... 8
6. HTTPS Connection to Web Application and REST Services .... 9
   6.1. Create a Keystore ................................ 9
   6.2. Block HTTP Connections to the Server ......... 11
   6.3. Invalidate Open Sessions ....................... 12
7. Certificate Installation .............................. 13
8. Password Storage ..................................... 15
   9.1. User Authentication and Authorization: REST Services .... 15
   9.2. User Authentication and Authorization: Web Application .... 18
10. Summary .............................................. 19
11. References .......................................... 19
12. Abbreviations ....................................... 20
1. Introduction

Smart environments refer to any physical world that is richly and invisibly equipped with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects and are continuously working to make inhabitants’ lives more comfortable [1]. Security is a fundamental issue in these systems especially when it involves private and confidential data about the environment or the users. One of the promising technologies used in smart environments is semantic technology. It is a way to enable data to be machine understandable. This way, data can be directly and indirectly processed by machines. To facilitate the development of such semantic smart environment applications, Smart-M3 architecture can be used. It is an open source architecture that consists of two main components: semantic information broker (SIB) and set of knowledge processors (KPs). SIB is the main storage for saving the application data semantically in RDF format. All the scenarios are then implemented through set of KPs that interact with the SIB to insert, update, or remove its data. Then finally the smart environment application can provide an interface to its end users through web or mobile applications that use set of exposed services to build meaningful and rich scenarios. The current implementations of Smart-M3 don’t natively support security issues like message encoding and access control.

The aim of this technical report is to introduce some mechanisms to provide such features to Smart-M3 based systems. Typical system architecture is considered that consists of the SIB as the core storage for the environment information, set of KPs that interact with the SIB and cooperate to achieve the required application scenarios, and set of REST services that are exposed to end user mobile or web applications.

The following mechanisms are introduced as workarounds to raise the level of security in Smart-M3 based systems:

- SIB access restrictions through enabling firewall
- Securing the communication channels between the SIB and the applications through using HTTPS protocol to interface to the web applications and REST services
- Certifying the Android based clients
- Encrypting user credentials, and applying user authentication and authorization polices to properly access system information and services.

The security mechanisms explained in this technical report were implemented in the Smart Advertising pLatform for Egypt (SALE) project which was developed as part of RECOCAPE EU project [2].

The rest of the technical report is organized as follows: Section 2 introduces the Smart-M3 architecture. Section 3 presents the system development
environment. Section 4 to Section 8 explains the workarounds proposed to secure the smart environment application. Section 9 gives a summary on the technical report. The references and abbreviations are listed in Sections 10 and 11 respectively.
2. Smart-M3 Architecture

Smart-M3 is an open source platform that aims to provide a semantic web information sharing infrastructure between software entities and devices [3][4]. One of its main goals is to enable smart environments. Applying semantic web technologies to the data originating from the environment enable the environment to be machine understandable and hence smart [1].

Smart-M3 consists of two main components: semantic information broker (SIB) and knowledge processors (KPs). The environment information is semantically stored in one or more SIBs [3]. In the simplest case, one SIB will store all information about the environment. Collaboration of KPs forms the application. The Smart Space Access Protocol (SSAP) is the protocol that the KPs use to access the SIB. The principles guiding the design of Smart-M3 are simplicity, extensibility and being agnostic to the used communication mechanisms.

The SIB is the access point for receiving information to be stored or retrieving such stored information. Figure 1 shows a typical architecture for Smart-M3 based systems. It consists of the Semantic Information Broker (SIB) as the core storage for the environment information, set of Knowledge Processors (KPs) that interact with the SIB and cooperate to achieve the required application scenarios, and set of REST services that are exposed to end user mobile or web applications.

The current implementations of Smart-M3 don't natively support security issues like message encoding and access control [5]. This is unacceptable in real-world environments that usually contain private and confidential data. Workarounds to raise the security level of such systems will be introduced in the following sections.

Figure 1. System architecture
3. System Development Environment

This section lists the tools used to develop the smart environment system upon which security workarounds discussed in this technical report are applied. Table 1 lists the tools along with their description.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Link</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smart-M3_B_v0.3.12.tar.gz</td>
<td>0.3.12</td>
<td><a href="http://sourceforge.net/projects/smart-m3/">http://sourceforge.net/projects/smart-m3/</a></td>
<td>Smart-M3 architecture</td>
</tr>
<tr>
<td>Sofia_KPICore_v6.2_(SRC)_+SPARQL.zip</td>
<td>6.3</td>
<td><a href="http://sourceforge.net/projects/smartm3-javakpi/">http://sourceforge.net/projects/smartm3-javakpi/</a></td>
<td>Java Knowledge Processor Interface (KPI); a library that implements the required SSAP operations</td>
</tr>
<tr>
<td>Java Development Kit (JDK)</td>
<td>1.6 or above</td>
<td><a href="http://www.oracle.com/technetwork/java/javase/downloads/index.html">http://www.oracle.com/technetwork/java/javase/downloads/index.html</a></td>
<td>Java development kit</td>
</tr>
<tr>
<td>Android SDK</td>
<td>API level 17 or above</td>
<td><a href="http://developer.android.com/sdk/index.html">http://developer.android.com/sdk/index.html</a></td>
<td>Android SDK</td>
</tr>
<tr>
<td>Tomcat Application Server</td>
<td>7.0 or above</td>
<td><a href="http://tomcat.apache.org/download-70.cgi">http://tomcat.apache.org/download-70.cgi</a></td>
<td>Tomcat server</td>
</tr>
</tbody>
</table>

4. Solution Strategies

In order to raise the level of security in Smart-M3 based systems, some workarounds has been proposed. Some of these workarounds are to be applied to the server side of the application while the others target the client side. Figure 2 summarizes the suggested mechanisms whose details are explained on the following sections.
5. SIB Access Restrictions – Set Firewall Restrictions

The SIB should only allow access from the web application or REST services, thus only those machine IP’s, MACs, and ports should be allowed. This can be done, e.g., in Ubuntu by iptables [6] or uncomplicated firewall [7]. These can also be used to make limits on requests from un-allowed addresses. Note that they cannot be used to protect against a real DOS attack, for example.

The following are examples for using iptables to restrict IP and MAC addresses in case SIB and web server are running on same or different machines:

**In case SIB and web server are on the same machine**

```shell
iptables -A INPUT -p tcp -s localhost --dport 10010 -j ACCEPT
iptables -A INPUT -p tcp --dport 10010 -j DROP
```

**In case SIB and web server are on different machines**

```shell
iptables -A INPUT -p tcp --d port server_ip_address --dport 10010 -j ACCEPT
iptables -A INPUT -p tcp --dport 10010 -j REJECT
```

**Whether SIB and web server are on same or different machines**

```shell
iptables -A INPUT -m mac --mac-source 00:0F:EA:91:04:08 -j ACCEPT
iptables -A INPUT -i eth1 -j DROP
```
The target REJECT means to drop the packets and send an error message to remote host. The target DROP means to drop the packet and do not send an error message to remote host [8].

Note: for development purposes you can limit the access to tomcat application server in tomcat/servers.xml as follows:

```
<Valve className="org.apache.catalina.valves.RemoteAddrValve"
allow="127.0.0.1\|192.168.3.144" />
```

6. HTTPS Connection to Web Application and REST Services

Hypertext Transfer Protocol Secure (HTTPS) is a communication protocol for secure communication over a computer network [9]. It is the result of simply layering the Hypertext Transfer Protocol (HTTP) on top of the SSL/TLS protocol, thus adding the security capabilities of SSL/TLS to standard HTTP communications.

We will create a SELF-SIGNED certificate for the web application. This means, that users have to accept the self-signed certificate in order to access the web application of the system from their browsers. This provided solution is for demonstration of the idea. A certificate should be purchased from a Certificate Authority [10] in case the system is in production phase for increased security and improved user experience. This way the web application could be trusted directly by Web browsers and mobile clients.

The following are the steps required for creating the SELF-SIGNED certificate:

6.1. Create a Keystore

If you have %JAVA_HOME% environment variable defined, then these commands should work directly from Windows Command Processor. If not, you can find the keytool.exe in java jdk/bin folder.

1) keytool -genkey -alias alias_for_the_key_store -keyalg RSA -keystore /home/administrator/salekeystore -ext san=dns:www.example.com or -ext san=ip:server_ip_address

   a. When asked your name first, enter localhost or the hostname used by the server.

   b. keystorePass should be sufficient and not trivial.

The option (-ext san=dns:www.example.com or -ext san=ip:server_ip_address) of the keytool command is basically defining the CN (common name). Hostname or IP, without it java clients would give an error "HTTPS hostname wrong: should be <xxx.xxx.xxx.xx> error message." and would not connect to the HTTPS address.
2) Modify `Tomcat\conf\server.xml` to:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
    maxThreads="150" scheme="https" secure="true"
    clientAuth="false" sslProtocol="TLS"
    keystoreFile="/home/administrator/salekeystore"
    keystorePass="keystore_password"/>
```

3) Restart tomcat.

Navigate to the main webpage of your web application. You will get similar page to that shown in Figure 3. Accept and save the self-signed certificate From “Add Exception” button in the bottom, you will be brought to the web page via HTTPS protocol. Figure 4 shows the web interface of the Smart Advertising pLatform for Egypt (SALE) that was developed as part of RECOCAPE EU project [2].

![Figure 3. Accessing the web application through HTTPS protocol](image-url)
6.2. Block HTTP Connections to the Server

Configurations in Section 6.1 allow every page to run both secure and insecure, thus we will now disable using the HTTP.

1) Modify Tomcat/conf/web.xml file, add the following snippet.

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>sale</web-resource-name>
    <url-pattern>/*</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

- `<url-pattern>`: defines what is restricted
- `<web-resource-name>`: here sale is the name of the constraint section; any name can be used and should work

2) Restart tomcat

3) Navigate to the webpage of your application using the HTTP protocol (e.g. http://10.130.201.80:8080/saleweb/ in case of SALE project).
4) You will be redirected to the HTTPS secured link (e.g. https://10.130.201.80:8443/saleweb/index.jsp in case of SALE project), i.e., only secure access is allowed.

Note: you could define different restrictions for the web application and the REST services, just make different url-patterns for both of them and define two security-constraints. We could also restrict HTTPS to login page, but for overall security, it is crucial to keep HTTPS on for every page.

```xml
<security-constraint>
  <web-resource-collection>
    <web-resource-name>sale</web-resource-name>
    <url-pattern>/adminHome.jsp</url-pattern>
  </web-resource-collection>
  <user-data-constraint>
    <transport-guarantee>CONFIDENTIAL</transport-guarantee>
  </user-data-constraint>
</security-constraint>
```

6.3. Invalidate Open Sessions

For security reasons, open sessions should be invalidated. For example, if an admin logged in to the web application, and left this session opened, then another user returns to the admin page through its URL, he can access the previously logged in user's information. Invalidating open sessions is done by adding logout.jsp servlet that simply run the following code shown in Figure 5. Sessions are to be invalidated when the user clicks the logout button or close the browser.

```java
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException {
    request.getSession().invalidate();
    response.sendRedirect("index.html");
}
```

Figure 5. logout.jsp sample code
7. Certificate Installation

In order for the mobile client to access the HTTPS REST service, we have to extract the server certificate (generated in Section 6.1) and export it to the mobile client. This is done on Android mobile client through the following steps:

1. Run the command:
   
   ```
   keytool -export -keystore salekeystore -alias "sale" -file key.cer
   ```
   
   - "sale" is an alias for the keystore, any alias can be used.

2. Create a keystore in bouncy castle format for Android by running the command:
   
   ```
   keytool -import -v -trustcacerts -alias "alias_for_the_key_store" -file key.cer -keystore androidkeystore.bks -storetype BKS -provider org.bouncycastle.jce.provider.BouncyCastleProvider -providerpath bcprov-jdk15on-1.46.jar -storepass keystore_password
   ```
   
   Note: You will need to download the providerpath `bcprov-jdk15on-146.jar` package from the internet. This jar file should be on the same directory where your exported `key.cer` file is located or the path to it, e.g., "-providerpath /home/administrator/bcprov-jdk15on-146.jar", if you are not on this folder when executing the command.

3. Check the certificate information and write y and press Enter.

4. Copy the `androidkeystore.bks` to `res/raw/` folder in the Android project, and create a key in `res/values/string.xml`

   ```
   <string name="store_pass"> keystore_password </string>
   ```

   In addition, the Android client source code should implement a class that enables access to the REST services through HTTPS instead of HTTP. This is done by using the previously generated keystore. Without it, we would have to bypass SSL warning/exceptions to work with SELF-SIGNED certificate, and disabling features makes it insecure. Figure 6 shows a snippet of the Android source code enabling the access to the REST services through HTTPS.
package com.secc.sale_rest_client;
import android.content.Context;

public class SecureHTTPClient extends DefaultHttpClient {

    private static Context appContext = null;
    private static HttpParams params = null;
    private static SchemeRegistry schemeReg = null;
    private static Scheme httpScheme = null;
    private static Scheme httpsScheme = null;
    private static String TAG = "SecureHTTPClient";

    public SecureHTTPClient() {
        if (httpScheme == null || httpsScheme == null) {
            httpScheme = new Scheme("http", PlainSocketFactory.getSocketFactory(), 8080);
            httpsScheme = new Scheme("https", mySSLSocketFactory(), 8443);
        }

        getConnectionManager().getSchemeRegistry().register(httpScheme);
        getConnectionManager().getSchemeRegistry().register(httpsScheme);
    }

    public SecureHTTPClient(Context myContext) {
        appContext = myContext;
        if (httpScheme == null || httpsScheme == null) {
            httpScheme = new Scheme("http", PlainSocketFactory.getSocketFactory(), 80);
            //httpsScheme = new Scheme("https", mySSLSocketFactory(), 443);
            httpsScheme = new Scheme("https", mySSLSocketFactory(), 8443);
        }

        getConnectionManager().getSchemeRegistry().register(httpScheme);
        getConnectionManager().getSchemeRegistry().register(httpsScheme);
    }

    private SSLSocketFactory mySSLSocketFactory() {
        SSLSocketFactory ret = null;
        try {
            final KeyStore ks = KeyStore.getInstance("BKS");
            final InputStream InputStream = appContext.getResources().openRawResource(R.raw.androidkeystore);
            ks.load(InputStream, appContext.getString(R.string.store_pass).toCharArray());
            InputStream.close();

            ret = new SSLSocketFactory(ks);
            System.out.println("ret:" + ret);
        } catch (UnrecoverableKeyException ex) {
            Log.d(TAG, ex.getMessage());
        } catch (KeyStoreException ex) {
            Log.d(TAG, ex.getMessage());
        } catch (KeyManagementException ex) {
            Log.d(TAG, ex.getMessage());
        } catch (NoSuchAlgorithmException ex) {
            Log.d(TAG, ex.getMessage());
        } catch (IOException ex) {
            Log.d(TAG, ex.getMessage());
        } catch (Exception ex) {
            Log.d(TAG, ex.getMessage());
        } finally {
            System.out.println("2ret:" + ret);
            return ret;
        }
    }
}

Figure 6. SecureHTTPClient class
In Android, you will define your HTTPClient to be an instance of SecureHTTPClient class as shown in Figure 7.

```java
HttpClient httpClient;
if(secure){
    httpClient = new SecureHTTPClient(appContext);
} else{
    httpClient = new DefaultHttpClient();
}
```

Figure 7. HTTPClient Instantiation

8. Password Storage

One of the main roles for security is not to store plaintext passwords anywhere, because if they are compromised they can be used directly to attack the user's services. Securing user's credentials can be done by sending them over secure line (HTTPS) and encrypting them so that they cannot be accessed by unwanted third parties. Digest and salt are used for this purpose. Instead of saving the password itself, the digest and salt will be saved in the SIB. Salt is basically random data added to the data that is used to form a hash from the password [11]. Digest is basically the result of combining password + salt into a hash value, i.e., one way transformation from original plaintext data to a cryptographic hash [12]. These are used to prevent replay attacks and to protect the user's password in the database.

9. User Authentication and Authorization

A KP is to be implemented to grant a unique username for each user who wants to use the system. Another KP will grant access to the system to every user already registered and that provides the right credentials. It then answers with the user's URI; an identifier assigned to each use upon creation to uniquely identify him. This approach simulates the control of a secured SIB implementation, where access control and user authentication are implemented internally. For demonstration purposes this approach can be used with an open SIB implementation for access control.

9.1. User Authentication and Authorization: REST Services

1. Modify Tomcat\conf\web.xml file, by adding the following snippet.
The AuthenticationFilter class is shown in Figure 8.

```java
package com.secc.webServiceAdaptationLayer;

import java.io.IOException;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Vector;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.ws.rs.core.Response;
import security.SecurePassword;
import com.secc.serviceAdaptationLayerKPs.UserAuthentication;
import com.sun.jersey.core.util.Base64;

public class AuthenticationFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain filter) throws IOException, ServletException {
        if (request instanceof HttpServletRequest) {
            // IF we want to, e.g., allow anonymous access to some parts of the service.
            // Naturally addUser has to be allowed without login.
            String pathInfo = ((HttpServletRequest) request).getPathInfo();
            List<String> listOfAllowed = Arrays.asList("/sale/addUserService",
            "/sale/loginService",
```
"/sale/sendMonitorInfo",
"/sale/getInterestDomainsService";
if(listOfAllowed.contains(pathInfo)){
  //bypass
  filter.doFilter(request, response);
  return;
}
// 1. check username and password
// 2. if these match, then put "authenticatedUser" == given user and later
// on can be used to check authorization
// save digest+salt in memory of the AuthenticationFilter - if used in every
// query should be saved?
// if in memory (user), check password
// else load from the sib and if successful authentication, save
// digest+salt in memory
String q_user = request.getParameter("user");
String q_pw = request.getParameter("password");
// NOTE: this is not a valid check for real system.
// For example, if user's are able to change their passwords, we'd have to
// update this cache at that event also
boolean authenticationOk = false;
Vector<String> digest = (Vector<String>) digests.get(q_user);
// no cache hit for digests
if(digest == null){
  digest = auth.getDigestAndSaltForUser(q_user);
  if(digest.size() != 0){
    digests.put(q_user, digest);
    }else{
    authenticationOk = false;
  }
}
// if valid
if(digest.size() == 2){
  boolean authenticated = auth.checkPassword(q_pw, digest.get(0),
    digest.get(1));
  if(authenticated){
    request.getServletContext().setAttribute( "authenticatedUser",
      q_user);
    authenticationOk = true;
  }else{
    authenticationOk = false;
  }
}else{
  authenticationOk = false;
}
count++;
if (response instanceof HttpServletResponse) {
  if(authenticationOk){
    //bypass
    filter.doFilter(request, response);
  }else{
    HttpServletResponse res = (HttpServletResponse) response;
    res.sendError(HttpServletResponse.SC_UNAUTHORIZED);
  }
  // If user's password was updated, remove it from the cache for next
  // query
  if(request.getServletContext().getAttribute("updatedUserPassword") !=
    null){
    digests.remove(q_user);
  }
  return;
}
In REST, we have to authenticate and authorize the user on every request, since REST is stateless. In its simplest form, username and password are sent as query parameters. The authentication filter extracts the username and password from the query. They are sent as plaintext, but should be secure since HTTPS is enforced. It should be noted that this could not be used without HTTPS as it would be easy to sniff the user’s password without secure line. Filter then checks the password against SIB contents, or in this case, the cached digest+salt of the user. It then adds an attribute to the request that is called “userAuthenticated” that indicates that the user has been authenticated.

Authorization is handled afterwards by adding other filters that will be called one after the other as a chain that can modify the data or restrict access to it.

9.2. User Authentication and Authorization: Web Application

In the web application, server sessions can be used and session authentication can then be saved. Authorization is checked with an AccessControlKP, which is a KP to be developed to check whether the user is authorized to access the information or not. It basically checks whether the defined user-authorizations are present in a given triple subject. E.g., N3 format triples: eg:ex eg:hasTime 125125155; eg:hasValue "Test".

eg:ex http://www.semanticweb.org/2013/Security#GetAllowedFor eg:User1. Then when user queries given value *, eg:hasValue, *, the KP checks whether there exists this GetAllowedFor our given user. The User has to be inserted into the KP per request basis, or per session basis, depending on whether KP is used for a longer period or shared via users.
10. Summary

Smart-M3 is one of the proposed platforms to develop smart environments. The core components of such systems are the SIB and the KPs. Application scenarios are realized through interactions between the KPs and the SIB to insert, update, or delete environmental or user data in the SIB. Set of services are then exposed to end user mobile or web applications, with REST services being the most common. Current realization of Smart-M3 architectures don’t natively support security features (being completely open) and thus leaving system information insecure and subject to attacks. This technical report introduced some mechanisms the aim at providing a good application level security despite the open or unsafe platform used. To achieve these results, firewall restrictions are set to limit access to the SIB. In addition, HTTPS is used instead of HTTP to provide access to the system from both web application and the REST services. Moreover, certificate is installed to Android based mobile clients to authenticate its access to the system. Authentication and authorization is supported to grant access to only intended users and intended services. And finally, credential information, like the passwords attached to the users are encrypted and are not saved as plaintext in the SIB, this makes them incomprehensible to third-party entities.

The security mechanisms explained in this technical report was implemented in the Smart Advertising pLatform for Egypt (SALE) project which was developed as part of RECOCAPE EU project.

All of the presented mechanisms contribute to raise the level of security of the system, yet they don’t guarantee the complete security when there is no real access control on the SIB itself. Some of the proposed mechanisms are common techniques for web apps and REST services, whilst many more techniques can be applied in that context. This report highlighted the most important techniques in the context of open Smart-M3 environments.

11. References


12. Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIB</td>
<td>Semantic Information Broker</td>
</tr>
<tr>
<td>KP</td>
<td>Knowledge Processor</td>
</tr>
<tr>
<td>KPI</td>
<td>Knowledge Processor Interface</td>
</tr>
<tr>
<td>RDF</td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>HTTPS</td>
<td>Hypertext Transfer Protocol Secure</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
</tbody>
</table>