Enhancing Bluetooth Service Discovery Protocol
Sasikanth Avancha, Anupam Joshi, Timothy Finin
Department of Computer Science and Electrical Engineering
University of Maryland Baltimore County
{savanchi,joshi,finin}@csee.umbc.edu

This technical report summarizes current work done in enhancing Bluetooth SDP to include semantic service matching to support more successful service discovery. A brief overview of the existing Bluetooth SDP [1] is followed by a discussion of its limitations. Finally, an overview of the enhanced SDP and details of its working are discussed.

1 Overview of Bluetooth SDP

Bluetooth SDP enables a client application on a device to discover services on other BT devices in RF range. Based on current specifications, it is clear that each service (or service class) is represented by a profile. Each profile and associated attribute is identified by a 16-bit or 32-bit Universally Unique IDentifiers (UUID). All search queries are based primarily on UUIDs. Every query is required to contain at least one UUID. Querying for search records based on arbitrary values is not provided; rather, service records must be searched only using UUIDs. The motivation for employing this constrained method of service discovery seems to be that UUID based matching would be very well suited to devices with a small form factor.

1.1 Problems with SDP

We believe that the kind of service discovery described above is very inflexible and geared towards highly resource constrained devices. Allowing only syntactic matching will generate simple “yes/no” type of responses that may not always be appropriate or useful. For example, current SDP forces the client to generate queries such as “Is the Printer profile supported?” followed by “Is duplex printing supported?” followed by “Is this a Postscript printer?” (all three queries could be sent together, but each must be associated with a UUID) and forces the server to match on UUIDs corresponding to “Printer Profile” and “duplex” and “color” attributes to send back “yes/no” type answers.

The second problem with Bluetooth SDP is that every client must first establish an L2CAP connection and then an SDP connection with the server before attempting to discover a service. If both the client and the server are slaves in a piconet, then the client must create a new piconet and become its master, while the server becomes a slave. Now there is an additional burden of switching roles on the client (slave in one piconet and master in another).

The third problem is the lack of a service registration mechanism in Bluetooth SDP. It is not clear whether this is a “non-requirement” or a “deferred requirement” of SDP as described in the specification. In purely ad-hoc situations registration of services may not be feasible, but it might be very relevant in fixed infrastructure situations. Consider a Bluetooth-enabled mall. It is certainly possible for stores in the mall to register their services with some central entity and for clients to discover them by simply querying the central entity.

1This work is funded by NSF grants CCR 0070802, IIS 9875433, and the DARPA DAML project
2 Reasons for Enhancing Bluetooth SDP

While the current SDP may be very well suited for small devices and simple services, it does not take into account the fact that ordinary desktops, laptops and powerful handhelds (like the Compaq iPAQ) could also be Bluetooth-enabled. Thus, for example, a Bluetooth-enabled iPAQ would certainly be capable of handling (in terms of memory, compute power etc.) complex queries generated by client applications and responding to such queries. SDP does not allow for the possibility that some or all servers are tethered while being Bluetooth-enabled so that piconets are actually formed by clients coming into RF proximity with the servers. In such a situation, an enhanced and powerful service discovery mechanism will certainly be a win.

2.1 Overview of Enhancements

2.1.1 Semantic Service Search

The core of the enhancements to the Bluetooth SDP consists of an XSB [4] based knowledge base and reasoning engine. In order to support the knowledge base and reasoning engine, enhancements have been made to the existing SDP server code. These include addition of a set of new request and response types (SDP_SEMANTICSEARCH_REQ_ONE, SDP_SEMANTICSEARCH_REQ_ALL, SDP_SEMANTICSEARCH_RSP, etc.). The working of the query-response mechanism of the enhanced SDP can be described in the following steps:

- Matching Server initialization code starts up and initializes XSB, loads the knowledge base and starts the reasoning engine.
- Client establishes an SDP connection to the Matching Server after device discovery.
- Client sends an SDP_SEMANTICSEARCH_REQ_ONE message to the Matching Server. The message contains, among other information like connection id, transaction id etc., an RDF query. For example, an RDF query describing a printer looks like this:

```xml
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:sdp_sim=""

<sdp_sim:Printer rdf:ID="Any">
  <sdp_sim:PrintColorQuality rdf:parseType="Resource">
    <sdp_sim:Priority>3</sdp_sim:Priority>
    <sdp_sim:Value>color</sdp_sim:Value>
  </sdp_sim:PrintColorQuality>
  <sdp_sim:PrintFileFormat rdf:parseType="Resource">
    <sdp_sim:Priority>6</sdp_sim:Priority>
    <sdp_sim:Value>Postscript</sdp_sim:Value>
  </sdp_sim:PrintFileFormat>
  <sdp_sim:PrintOutputFormat rdf:parseType="Resource">
    <sdp_sim:Priority>7</sdp_sim:Priority>
    <sdp_sim:Value>Letter</sdp_sim:Value>
  </sdp_sim:PrintOutputFormat>
  <sdp_sim:PrintResolution rdf:parseType="Resource">
    <sdp_sim:Priority>4</sdp_sim:Priority>
    <sdp_sim:Value>1440</sdp_sim:Value>
```
We note here that RDF [2] is being substituted by DAML [3] and a few changes can be made to the reasoning engine to include support to reason about DAML statements.

- Matching Server receives the message, parses it and determines it to be an SDP_SEMANTICSEARCH_REQ.

- Getting down a more specific level in the Matching Server, the function to process the semantic search request is called. This function creates a temporary file and writes the query into it. It also creates an output file name for the reasoning engine to write its results out to. Finally, it passes this information to another function that actually handles the query.

- The query handling function makes two XSB function calls in order to handle the query. The first is used to parse the RDF data into triples. The format of the triple used by this reasoning engine is (Predicate, Subject, Object). For example, the PrintColorQuality attribute shown above is parsed into the following triples:

```
rdftriple(priority,'#genid1',3).
rdftriple(value,'#genid1','color').
rdftriple('printcolorquality','any','#genid1').
```

The second XSB function invokes the XSB predicate in the reasoning engine to perform the semantic search.

- It is not necessary for the RDF query to be complete (as shown above); the query may only contain attributes and no associated values or priorities. The reasoning engine has been designed to handle all possible forms of RDF queries.

Assuming that the service requested was validated, each triple is read in from the temporary file and the attribute along with its corresponding value is extracted. A list of all attributes, values and corresponding priorities is created. The priority information, as shown above, is part of the ontology. It is assumed that attributes of all instances of a particular service class on the Matching Server have the same priority assigned to them. Queries specify attribute priority in addition to the value to match on. Thus, the basic decision making on attribute priority is with the user. However, as mentioned already, it is possible for the client to specify only the attribute name and nothing else in the query. Essentially attribute priority and attribute value are parameters that decide the ordering of attributes in the list. As expected, there are four possible results that decide this ordering. Table 1 summarizes these possibilities from the Matching Server’s point of view of whether or not the priority and/or the value have been specified by the Client.

The list formed after this decision making process is then sorted by priority. The idea of the above exercise is that the service instance that provides a closest match to the value of the highest priority attribute in the sorted list will be used to return attribute values for the remaining attributes requested. For each valid attribute of the service, the desired value and the priority are compared to the corresponding value(s) and priority in the KB. If there is an exact match on the highest priority attribute, the attribute
values from the appropriate instance are written to the output file. Finally, the contact URI [5] for the service instance is determined and written to the output file. This is the response to the query. If there is no match or if the Client did not specify a value, the reasoner either picks the first available instance or all available instances based on whether this was an SDP_SEMANTICSEARCH_REQ.ONE or SDP_SEMANTICSEARCH_REQ.ALL request, respectively, and returns corresponding attribute values and the contact URI.

Finally, it is also possible for the client to specify only the service name and nothing else in its query. In this case, the response contains value of all direct attributes (not inherited) of all instances of the service and the contact URI for each instance.

### 2.1.2 Service Registration

Service registration is accomplished by the service provider via a SDP_SEMANTICSERVICEREG_REQ to a Registration Server. This request consists of three parts:

- **Service instance UID:** The service instance to be registered must be unambiguously identifiable on a given Registration Server. This implies that the service instance identifier must be unique. The best source of this identifier would be the “Contact URI” of the service instance. It is reasonable to expect that the “Contact URI” would be unique because any Client that discovers the corresponding service instance must be able to unambiguously access and use it. We propose that the “Contact URI” could be used as the UID directly or after passing through a simple Universal Hash Function.

- **Lease Time:** The service instance can be valid at the Registration Server for a fixed amount of time. The service provider is expected to renew this lease via SDP_SERVICELEASERENEW_REQ before it expires. If no renewal request is sent, the Registration Server unregisters the instance.

- **Service Instance description:** An RDF description of the service instance.

The following steps summarize the registration process.

- The UID and lease time are extracted and stored on disk.

- The RDF description is parsed and the triples are loaded into the KB.

- If it does not exist, a process to check for lease time expiration is created. This process is alive as long as the SDP Registration Server is alive. This process decrements the lease time periodically and when the lease time is down to 0, it unloads the corresponding triples from the KB (using the UID).

- It is possible for the service provider to renew the lease by sending the UID and a new lease time in a SDP_SERVICELEASERENEW_REQ. The service provider may also cancel the service via the SDP_SERVICELEASESECANCEL_REQ.
3 Status of this Work

This work is currently in progress. Experiments with different scenarios involving multiple Bluetooth nodes will be carried out. Results of these experiments will be reported in other publications including the M.S. thesis of Sasikanth Avancha.

References


