QoS-aware Self-adaptation of Communication Protocols in a Pervasive Service Middleware

Weishan Zhang\(^1\) Klaus Marius Hansen\(^1\) João Fernandes\(^1\) Julian Schütte\(^2\)
Francisco Milagro Lardies\(^3\)
\(^1\)Department of Computer Science, University of Aarhus, Aabogade 34, 8200 Århus N, Denmark
\(^2\)Fraunhofer Institute for Secure Information Technology, Rheinstrasse 75, 64295 Darmstadt, Germany
\(^3\)Telefonica I+D, PT Walqa Ed 1., 22197 Huesca, Spain
\{zhangws, marius, jfmf\}@cs.au.dk julian.schuette@sit.fraunhofer.de faml@tid.es

Abstract

Pervasive computing is characterized by heterogeneous devices that usually have scarce resources requiring optimized usage. These devices may use different communication protocols which can be switched at runtime. As different communication protocols have different quality of service (QoS) properties, this motivates optimized self-adaptation of protocols for all devices, considering power consumption and other QoS requirements, e.g. round trip time (RTT) for service invocations, throughput, and reliability. In this paper, we present an extensible approach for self-adaptation of communication protocols for pervasive web services, where protocols are designed as reusable connectors and our middleware infrastructure can hide the complexity of using different communication protocols to the upper layers. We also propose to use Genetic Algorithms (GAs) to find optimized configurations at runtime to achieve self-adaptation of web service transport protocols (TCP, UDP and Bluetooth), taking into consideration of all QoS requirements. Our tests show that protocol switching involves minimum performance overhead and runs efficiently. Our evaluations also show that the proposed approach for achieving self-adaptation for communication protocols is effective where optimized configurations of protocols can be obtained with acceptable performance and quality by GAs.

1 Introduction

Pervasive computing systems [1] are characterized by heterogeneous devices with different resources and communication capabilities, where wired connection, WiFi\(^1\), Bluetooth (BT)\(^2\), or Zigbee\(^3\) may be used. Some devices may be powered by external electricity, and some are running on batteries. Despite the power supply a device can have, it is always important to run services efficiently in terms of power consumption, latency, and reliability. These parameters are dependent on the communication protocol(s) used between devices as different protocols have different QoS (Quality of Service) properties. Therefore, QoS is an important factor when providing services to a user, in which the communication protocols used play an important role [2].

A large number of pervasive computing devices are equipped with multiple communication capabilities, which then can be utilized to adapt for different QoS requirements at runtime. This is an important feature of self-adaptation for self-management [3] in pervasive systems. Self-management includes a rich set of features, such as self-configuration, self-adaptation, self-optimization, self-protection and self-healing. We have reported our former work on self-healing [4], self-configuration [5] and self-protection [6]; we will concentrate on QoS-aware (including the awareness of power consumption) self-adaptation of communication protocols in this paper.

There has been some work on designing new adaptive communication protocols for pervasive computing [7], and research on testing the performance of existing protocols [8][9]. Because devices on the market most probably will not support new designed protocols, it is equally important, if not more, to make full use of the existing protocols in a self-adaptive manner, i.e., to use different protocols on a device for different QoS requirements dynamically. This is especially important for pervasive service computing, where services are designed to integrate existing devices and applications. To the best of our knowledge, there is very scarce

\(^1\)http://www.wi-fi.org/
\(^2\)http://www.bluetooth.com/
\(^3\)http://www.zigbee.org/
work in this area.

In this paper, we will first quantify the performance and power consumption for pervasive web service (SOAP\textsuperscript{4}) based) transportation using TCP, UDP and BT, which then will be utilized as context for achieving optimized self-adaption of these protocols. We explore the self-adaptation in a pervasive web service middleware called Hydra\textsuperscript{5}, which uses different transport protocols (currently TCP, UDP and BT) to transport web services calls and responses. These protocols are designed as reusable software connectors and the Hydra middleware infrastructure can hide the complexity of using these different communication protocols, which can be switched at runtime. To make a pervasive system running in a globally optimized manner, we also propose to use multi-objective optimization using Genetic Algorithms (GAs) to find optimized configurations at runtime, to achieve transportation protocol self-adaptation taking into consideration of power consumption and other QoS requirements. The tests and evaluations show that our approach is effective.

The rest of the paper is structured as follows: Section 2 presents a scenario for self-adaption usage. Then in Section 3 we present our measurements for QoS parameters for different platforms and protocols. The design and implementation of the infrastructure support for communication protocol self-adaptation are shown in Section 4. In Section 5 we present the design of GAs to find optimized protocol configurations at runtime. Section 6 presents the tests of performance overhead, and the performance for obtaining the optimized configurations using GAs. We compare our work with the related work in Section 7. Conclusions and future work end the paper.

2 A scenario for self-adaption of communication protocols

In the following scenario, device services are using web service proxies for the sensor/actuator devices running JSE platform. Communication between devices can use TCP, UDP, and Bluetooth to transport pervasive web services calls and responses.

John owns a SmartHome system, which is equipped with self-managed heating, ventilating and home surveillance subsystems. While he is abroad for an academic conference, his home security system automatically turns on to the highest security level when detecting he is far away from home, and the whole SmartHome system is self-configured and self-adapted to minimize power consumption. All the devices used for the SmartHome system are equipped with both WiFi and Bluetooth support. These devices include 5 surveillance IP cameras, 10 motion detectors, 10 sound detectors, 10 ventilators and 10 heating controllers, and 10 thermometers. The SmartHome also has a management controller running on a gateway device that connects to the Internet either using WiFi or wired connection. This device is also able to communicate using Bluetooth. John has a mobile phone with him and can use it to check that everything is fine at home, this includes retrieving captured pictures from all surveillance IP cameras within one seconds, and reading the ventilating and heating conditions to make sure that his cat is living a comfortable life while he is away.

Our experiments [2] show that different communication protocols have different QoS properties, in terms of power consumption, throughput, reliability, etc. Here, reliability is defined as the number of successful calls divided by the total number of calls. It is also obvious that the local optimal configurations of protocols are not necessarily globally optimal. In the above scenario, to save energy, it seems natural to use Bluetooth for all devices. But this choice may not meet reliability and/or latency requirements for getting information on the status of the SmartHome system. The objectives for high reliability, low power consumption, low latency are conflicting: from our experiments, Bluetooth has low power consumption but also a high latency. Therefore, we need to find a global optimization for configuring the underlying system at runtime to meet all QoS requirements.

3 Quantifying power consumption and performance for transport protocols

We have tested the power consumption and performance for BT 1.2/2.0, WiFi (TCP/UDP), and wired network to transport pervasive web services, on mobile phones (Nokia 6630, Nokia N95) and laptops (Thinkpad T61p, T61) running JSE/JME/OSGi platforms. The Nokia Energy Profiler\textsuperscript{6} and a WaveTek Meterman 38XR\textsuperscript{7} were used to retrieve raw data for measuring power consumption. Because of space limits, the details on test procedures and other equipments used will not be shown here but can be referred in [2]. We list some test data in Table 1. Some data (BT2.0 to BT1.2, W-TCP to W-TCP and W-UDP to W-UDP) are going to be used in this paper ("W-TCP" stands for TCP communication using WiFi, "W-UDP" stands for UDP communication using WiFi, "BT" stands for Bluetooth, UDP stands for UDP communication on wired network).

From these tests, we can see that the OSGi based implementation of a web service has more or less the same performance and power consumption as the JSE based implementation. We can also see TCP connections have best

\textsuperscript{4}http://www.w3.org/TR/soap/
\textsuperscript{5}http://www.hydramiddleware.eu/
\textsuperscript{6}http://forum.nokia.com/info/sw.nokia.com/id/324866ec-0460-4fa4-ac53-018c392d4f0f/Nokia_Energy_Profiler.html
\textsuperscript{7}http://www.tequipment.net/Wavetek38XR.html
reliability as 100%, worse throughput and RTT than UDP, and obviously, higher power consumption than Bluetooth.

These QoS properties are parts of context for self-management, encoded in a QoS ontology (Figure 1), as one ontology in a set of self-management ontologies called SeMaPS ontologies, which are used in Hydra to achieve context-awareness based self-management [10][11], including self-adaptation. The QoS ontology contains the concepts and instances for QoS properties that are interesting to Hydra, most importantly for this paper: PowerDrain is used to model the power consumption, RTT time is used to model the round trip time, ResponseTime is used to model the execution time for a service, and Reliability is used to define the percentage of successful calls with respect to the total calls for a connection, also Reliability models a service’s capability to answer calls all the time. Figure 1 shows these concepts and some of the instances. The provision of the QoS context is realized as SQWRL\(^8\) queries, as presented in the former work [6].

### 4 Infrastructure support for switching communication protocols

To make the self-adaption of communication (transport) protocols possible, these protocols must be reusable and changeable at runtime. At the same time, as we are targeting pervasive web service environments, the underlying infrastructure must be able to know the changes of protocol, as well the new locations of the services. With this information all middleware components can perform necessary changes and hence hide the underlying protocol changes to the upper layers, where end users will not notice these changes. In the next section, we will show how we handle these issues by implementing protocols as software connectors, enhancing services to be capable of protocol switching, and also designing infrastructure to support the corresponding changes resulted from protocol switching.

\(^8\)http://protege.cim3.net/cgi-bin/wiki.pl?SQWRL

---

### Table 1. Part of QoS properties

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
<th>Net.</th>
<th>Plat.</th>
<th>RTT (ms)</th>
<th>Power (W)</th>
<th>Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>BT1.2</td>
<td>JSE</td>
<td>BT2.0</td>
<td>JSE</td>
<td>39.76</td>
<td>0.183(C)</td>
<td>97.29%</td>
</tr>
<tr>
<td>BT2.0</td>
<td>JSE</td>
<td>BT2.0</td>
<td>JSE</td>
<td>70.1</td>
<td>0.041(S)</td>
<td>100%</td>
</tr>
<tr>
<td>BT2.0</td>
<td>JSE</td>
<td>BT2.0</td>
<td>JSE</td>
<td>193.39</td>
<td>0.285(S)</td>
<td>100%</td>
</tr>
<tr>
<td>W-TCP</td>
<td>JSE</td>
<td>W-TCP</td>
<td>JSE</td>
<td>25.99</td>
<td>1.360(S)</td>
<td>100%</td>
</tr>
<tr>
<td>UDP</td>
<td>JSE</td>
<td>W-UDP</td>
<td>JSE</td>
<td>8.16</td>
<td>1.361(S)</td>
<td>99.93%</td>
</tr>
<tr>
<td>W-UDP</td>
<td>JSE</td>
<td>W-UDP</td>
<td>JSE</td>
<td>7.5</td>
<td>1.361(S)</td>
<td>99.90%</td>
</tr>
</tbody>
</table>

"Net." stands for network, "Plat." stands for platform, "Rel." stands for reliability; "(S)" stands for power consumption for a server side, "(C)" stands for power consumption for a client side.
To facilitate the usage of different protocols for pervasive service, we are using the pervasive web service compiler Limbo [13, 14] to generate web service code that can use different transport protocols, e.g., the developer can generate web services that use TCP, UDP and/or Bluetooth as transport protocol(s). The behaviors of these protocols are realized by the operations mentioned above, where the protocols have to implement a ServerProtocol interface for server sides and a ClientProtocol interface for client sides.

The design of protocols as connectors is extensible. If a device supports a protocol other than TCP/UDP and Bluetooth (these are currently specified in Limbo), it can be easily supported by adding the protocol to the Limbo compiler, through an implementation of the ClientProtocol and ServerProtocol interfaces (ref. Figure 2).

### 4.2 Positioning services for self-adaptation of communication protocols

In order to be capable of switching communication protocols, a device has to provide a Web Service interface that allows an external component to trigger the switching of protocols. This service is part of self-management services and it provides, in general, the following basic operations:

- **String changeProtocol(String newProtocol):** Performs a protocol change for a service the device is currently providing. The new protocol can be “TCP-Protocol”, "UDPProtocol" or "BTProtocol". This method returns the name of a service that has changed its protocol and the new endpoint of that service. For example after changing to Bluetooth protocol for a thermometer, you may get the following end point for a thermometer service: `ThermometerService=btspp://001C26E6D5E6:1;name=th03`. And of course, you can change protocols to all the services a device is currently providing at the same time, as shown in the following list for services provided by a thermometer after changing to Bluetooth.

```plaintext
ThermometerService=btspp://001C26E6D5E6:1;name=th03
SelfManagementService=btspp://001C26E6D5E6:1;name=SelfManagement
HydraService=btspp://001C26E6D5E6:1;name=hydr
```

- **String getSupportedProtocols():** Returns all the supported protocols of the device.
- **String getCurrentProtocol():** Returns the currently used protocol.

Figure 2 shows the details of relationships of the server side protocols, and the corresponding operations, using a thermometer (called TH03) service that provides getTemperature and getStatus operations. Two threads are used to achieve the separation of handling different concerns: one is used for the handling of actual device services, and the other one is dedicated for the handling of protocol switching.

### 4.3 Support for communication protocol switching with SOAP tunneling

The Network Manager and SOAP Tunneling [15] components of the Hydra middleware build an overlay P2P network [16] and provide mechanisms for service registration, discovery and consumption over this P2P network through different communication protocols, such as TCP, UDP or BT, allowing devices to offer and consume services in a Web Service compliant form. For registering a service in the Network Manager, the device requests the creation of an identifier (called HID – Hydra Identifier), providing the real endpoint of the service. The HID allows the service to be uniquely addressed and consumed over the Hydra overlay network.

The real endpoint of the service is used by the Network Manager for the last mile service invocation. This endpoint represents the actual location where the service is deployed and is, in contrast to the HID address, protocol-specific. When a service invocation arrives to a Network Manager from the P2P network, the SOAP Tunneling component forwards the service request to the correct endpoint with which that service was previously registered.

In Hydra, the protocol switching service of a device is called by the Self-Management component [11, 6]. In case of protocol changes, the Self-Management component will notify the Service Discovery Manager that the device’s services are now available in different endpoints. In order to achieve this the Self-Management component makes use of the Hydra Event Manager which uses a topic based publish-subscribe style [17], by publishing an event with topic "protocolChange" that the Service Discovery Manager is a subscriber to. After receiving this event, the Device Discovery Manager needs to update the information about the device’s services in the Hydra Network Manager, by calling the method renewHIDInfo(String hid, String endpoint), providing the new endpoint on which the service will be addressable. The device information (endpoints) is now updated in the Hydra Network and the services can now be addressed at their new endpoints. This is shown in Figure 3.

### 4.4 Policy-based triggering of self-adaptation

Self-adaptation may either become necessary at a regular time interval or may be triggered by changes of the current situation. For example, when new devices join the network, a re-configuration of all network protocols can reduce the overall power consumption, or QoS requirements changed and new configurations are needed as in our scenario in Sec-
For triggering self-adaptation actions such as a protocol switching, we use Complex Event Streaming (CEP) to recognize event patterns of interest and event-condition-action policies to trigger the actual changes. The CEP engine retrieves events from the Hydra Event Manager (e.g., raw sensor data or network events), and aggregates them to semantically rich event patterns. Conditions are specified in Prolog and can refer to triplets from the Hydra self-management ontologies while actions are realized by Java classes implementing an Action interface. If the event pattern is triggered and the condition is evaluated to true, an action is invoked, sending a reconfiguration request to the Self-Management component, which will first use GAs to obtain an optimized configuration of communication protocols used by all channels, and make the corresponding changes as necessary to all protocols.

5 GAs based optimization of service transportation

Figure 4 shows the topology of devices connections for the scenario introduced in Section 2, which is of a typical star pattern. All sensor/actuator devices are connected to a gateway node, and this gateway node connects to the Internet. For this simple topology, there are 3^5 possible configurations of communication protocols (referring to TCP/UDP/BT), a scale that is well suited for genetic algorithms to find optimal solutions. Here we are not considering the link from the Gateway to the Internet, as it should use TCP/UDP as always, and from Table 2, the power con-
sumption, the gateway service execution time and its service reliability depend on the channels connected from these 55 devices to the gateway.

Consider the power that camera needs to take picture/video, heating control and other functioning consumption, we present the power consumption for all devices used in our scenario in Table 2 based on the data in Table 1. These power consumption values reflect a device that is in operation mode. We also show the service execution time on devices and also the service reliability which are the same for a device irrespective of the protocols used (except the gateway).

Figure 4. Topology of the example scenario

Table 2. QoS data used for our scenario

<table>
<thead>
<tr>
<th>W-TCP</th>
<th>W-UDP</th>
<th>BT</th>
<th>Power</th>
<th>E-Time</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gateway</td>
<td>X</td>
<td>X</td>
<td>2.9</td>
<td>360</td>
<td>99.993%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>2.7</td>
<td>346</td>
<td>99.999%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>2.6</td>
<td>356</td>
<td>99.993%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.3</td>
<td>364</td>
<td>99.93%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>2.5</td>
<td>348</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Camera</td>
<td>X</td>
<td>X</td>
<td>0.5</td>
<td>230</td>
<td>99.990%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.7</td>
<td>1.7</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Motion</td>
<td>X</td>
<td>X</td>
<td>0.3</td>
<td>200</td>
<td>99.9999%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.5</td>
<td>1.5</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Sound</td>
<td>X</td>
<td>X</td>
<td>0.3</td>
<td>200</td>
<td>99.9999%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.5</td>
<td>1.5</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Thermometer</td>
<td>X</td>
<td>X</td>
<td>0.3</td>
<td>210</td>
<td>99.9999%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.5</td>
<td>1.5</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Ventilator</td>
<td>X</td>
<td>X</td>
<td>0.7</td>
<td>240</td>
<td>99.993%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.9</td>
<td>1.9</td>
<td>99.9999%</td>
</tr>
<tr>
<td>Heating</td>
<td>X</td>
<td>X</td>
<td>0.6</td>
<td>260</td>
<td>99.998%</td>
</tr>
<tr>
<td></td>
<td>X</td>
<td>X</td>
<td>1.8</td>
<td>1.8</td>
<td></td>
</tr>
</tbody>
</table>

"X" stands for that protocol will be used. "E-Time" stands for service execution time (response time).

We have explored using GAs for selecting optimized self-configuration [5] and self-protection [6]. The self-adaptation problem has a much bigger problem space than the former work (e.g. \(3^{10} - 1\) times bigger than that of the self-protection problem presented in [6]).

5.1 Optimization objectives and constraints formulation in GAs

The power consumption of a system (the \(P\) objective) is calculated by the sum of each node’s power consumption as:

\[
P = \sum_{i=1}^{n} \sum_{j=1}^{m} P_i \cdot C(i, j) \tag{1}\]

Here \(C(i, j) = 1\) if a communication protocol \(j\) is used for node \(i\) (with a scope of \([1, n]\)) that has power consumption \(P_i\) is selected, otherwise \(C(i, j) = 0\). \(j\) represents the sequence number of a concrete communication protocol with a scope of \([1, m]\). In our scenario, \(n = 55\) and \(m = 3\).

The latency will be considered for the channels created between the sensor/actuator devices and the gateway node. It is calculated as the maximum of the latency from these channels plus the gateway service execution time which depends on the overall protocol configurations as shown in Table 2. Therefore the latency (the \(L\) objective) of the system under consideration is calculated as:

\[
L = \max((L_{11} + T_1, L_{12} + T_2, \ldots, L_{1n} + T_n, \ldots, L_{mK} + T_K)) \cdot C(i, j) + L_g \tag{2}\]

Here \(C(i, j) = 1\) if a communication protocol \(j\) is used for channel \(i\) (with a scope of \([1, K]\)) that has latency of \(L_i\) is selected, otherwise \(C(i, j) = 0\). Here \(i\) also denotes the sequence number of a device that is connected to the gateway node, where \(T_i\) represents the service execution time (ResponseTime in Figure 1). \(j\) represents the sequence number of a concrete communication protocol with a scope of \([1, m]\). In our case, there are 55 channels if not considering the one where the gateway connects to the Internet. \(L_g\) represents the service execution time on the gateway, which depends on what are the protocols used as shown in Table 2.

The reliability can be calculated in a similar way as the latency objective. In order to uniformly minimize all three objectives, we calculate the un-reliability (the \(U_r\) objective) instead as follows, considering of the topology of our system:

\[
U_r = 1 - \min((R_1 \cdot S_1, R_2 \cdot S_2, \ldots, R_i \cdot S_i, \ldots, R_K \cdot S_K) \cdot C(i, j) \cdot R_g) \tag{3}\]

Here \(C(i, j) = 1\) if a communication protocol \(j\) is used for channel \(i\) (with a scope of \([1, K]\)) that has reliability of \(R_i\) is selected, otherwise \(C(i, j) = 0\). Here \(i\) also denotes the sequence number of a service running on a device that is connected to the gateway node, where \(S_i\) represents the service reliability. \(j\) represents the sequence number of a concrete communication protocol with a scope of \([1, m]\). \(R_g\) represents the service reliability on the gateway, which depends on what are the protocols used as shown in Table 2.

We also have restrictions for the overall system’s communication configurations (referring to all 55 channels):
- **Reliability** should be maximized (highest value is 1 for a channel)
- **Latency** must not exceed 700 ms (leave 300 ms for the connection from the gateway to John through Internet)
- **Power** operation power consumption should be minimized, not more than 50 Watts for the whole system

5.2 Chromosome encoding and fitness evaluations

A chromosome in a GA stands for a unique solution in the solution space. GAs can typically make use of booleans, real numbers and integers to encode a chromosome. For our case, as explored in self-configuration [5] and self-adaptation [6], we represent a chromosome using integers (starting from 0). More specifically, an integer vector is used to represent a solution: \( V = [V_1, V_2, ..., V_i, ..., V_n] \) (where \( n \) is the number of decision variables – in our case 55) to represent a solution found by a GA. \( V_i \) is a natural number, acts as a pointer to the index of the communication protocol of the \( i \)th protocol. For example, a chromosome \([0,1,2,1,2,0,1,1,2,...]\) represents a solution which chooses Bluetooth for channel 1, W-TCP for channel 2, W-UDP for channel 3, and so on. Based on the chosen protocols, the GAs then decide fitness using the introduced objective equations, and will at the same time evaluate constraints mentioned in Section 5.1.

5.3 Pareto front for the self-adaptation problem

As recommended in our evaluation of GAs’ usability in self-management planning [5], we usage multiple GAs, including NSGA-II [18], MOCell [19], and FastPGA [20] (in which FastPGA is a new GA added to our Self-Management component) to find the Pareto front for our problem. A very interesting thing is that for our problem there are only four points in the front, that is to say we have a very big problem space, but very small solution space instead. This makes the further investigation of the GAs usage for our problem worth as it is a different case with our former exploration. The time consuming process of getting a Pareto front is not necessary, if you do not want to evaluate solution quality but just use our recommended parameters [5] for finding optimized configurations.

6 Evaluation and Performance

To make sure that the protocol switching during the self-adaptation process will not incur un-affordable overhead, we will first test the corresponding performance overhead in the service side. Second, we need to evaluate the performance to find optimized configurations of protocols, and investigate whether we have new recommendations of parameters settings for the new problem.

### 6.1 Performance overhead for switching protocols

We have used the thermometer web service to test the performance of protocol switching, with a continuous chain of protocol changes from BT to TCP, then to UDP, then back to BT, and then to UDP, and so on. We measure the time (in milliseconds) taken to finish a switch. A number of tests have been conducted and part of tests is shown in Table 4. We can see that the overhead for switching back and forth to TCP and UDP is more or less 0 ms (occasionally it takes 16 ms to be prepared), and the switching to Bluetooth takes 81 ms in average, because of the preparation of the device wireless unit. In general, this is quite acceptable in pervasive web services environment, especially considering that it takes usually more than 200 ms to answer a web service call.

### 6.2 Performance of finding optimized protocols configuration

We have extensively tested the performance for all the three GAs for the self-adaptation problem, following the parameter settings as recommended [5], and with extra tests where max evaluations are lower (2000, 3000, 40000) considering the bigger problem space, and also we tested cross over probability of 1. We now show a part of these tests in Table 5. As we noticed, although MOCell runs

### Table 3. Pareto front for the self-adaptation problem

<table>
<thead>
<tr>
<th>L objective</th>
<th>P objective</th>
<th>Ur objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>686.1</td>
<td>27.6</td>
<td>0.000170009592771581</td>
</tr>
<tr>
<td>694.1</td>
<td>25</td>
<td>0.000799946582317412</td>
</tr>
<tr>
<td>666.1</td>
<td>44.4</td>
<td>0.000170009592771581</td>
</tr>
<tr>
<td>670.1</td>
<td>43.5</td>
<td>0.00108992285954035</td>
</tr>
</tbody>
</table>

### Table 4. Overhead of protocol switching

<table>
<thead>
<tr>
<th>UDP to BT</th>
<th>to UDP</th>
<th>to TCP</th>
<th>To BT</th>
<th>to UDP</th>
<th>to TCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td>0</td>
<td>0</td>
<td>31</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>93</td>
<td>0</td>
<td>0</td>
<td>62</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0</td>
<td>109</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>109</td>
<td>0</td>
<td>0</td>
<td>109</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>78</td>
<td>0</td>
<td>0</td>
<td>109</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

faster than NSGA-II and FastPGA, in a normal setting (for example, cross over probability equals to 1, and population size 80 and maximum evaluations 3000), NSGA-II gets much higher quality of solutions than that of MOCell and FastPGA, where FastPGA performs the worst in terms of Hyper Volume quality indicator [21]. FastPGA [20] is normally used for computationally heavy situations, therefore as expected it did not show any advantages over other two GAs as our problem does not involve such operations. Because the problem space is bigger that in self-configuration [5] and self-protection [6], it takes longer time for both MOCell and NSGA-II (e.g. 1294 ms vs. 357 ms) to find optimized solutions which is reasonable. We can also see that the cross over probability of 1 for NSGA-II obtain better quality of solutions than other setting, but without significant increase of the running time needed, therefore we would now recommend to use this in the future.

### 6.3 Discussion

For a larger problem space, we would expect that for the same parameter setting, GAs will run slower, and this is consistent with what we show in Table 5. From the new tests as in Table 5, we recommended that for NSGA-II, which remains the best algorithm for all the tested self-management problems. As consistent with our former recommendations in [5] and [6], population size of 64-100 is recommended for NSGA-II (cross over probability should be 1 instead - we did not test this setting before). For a bigger problem space as in our case, the max evaluations should be lowered down to 2000 to 3000, in which we will get performance in between 412 ms to 642 ms, where average Hyper volume will be 0.179 to 0.188.

There may be very few points in a Pareto front for a practical self-management problem from our experience. For example, in a first attempt to optimize the self-adaptation configurations, we did not consider the reliability and execution time in the gateway node (ref. Table 2). We found that there are only two points that formed the Pareto front (using all three GAs but there probably some points can not be found with our two days running of these GAs). For only two points, the Hyper volume quality indicator is not usable, but this does not affect the usage of GAs to find optimized solutions using the recommended parameter settings.

### 7 Related work

As previously mentioned, there has been focus on creating new protocols for pervasive computing [22, 7]. Barr et al. [22] used the AODV protocol [23] for ad hoc routing. Wu et al. [7] presents a variant of a quorum protocol where mobile stations cycle through wake-up and sleep modes to increase energy efficiency. Our work differs in that we assume the existence of a set of available protocols out of which we need to choose among others to optimize energy usage and also other QoS properties at runtime. But this work will be considered in our approach as part of our future work to consider the dynamism of power efficiency.

Pervasive computing has always been a motivation for self-management and autonomic computing in particular [3] using a variety of approaches. There are explorations for using auction algorithms to dynamically resolve policy conflicts during context changes in the CARISMA middleware [24]. Poladian et al [25] used an analytical approach similar to combinatorial auctions in project Aura to adaptively support everyday tasks of users. We are also working on including this in the self-management planning layer, as auction algorithms can find local optimal if required. But in general, from our experiences of preliminary tests, we will still use GAs as the main approach for finding global optimized solutions as it can find better quality of solutions with reasonable performance than the auction algorithms.

Genetic algorithms are being used in self-managing systems in which configurations are encoded as utility functions and where the problem of finding a (Pareto) optimal configuration becomes a multi-objective optimization problem [26, 27]. A recent approach within pervasive computing is the MUSIC middleware[28]. In MUSIC, components have QoS characteristics and the assumption is that QoS of a composition can be computed. In contrast to our work, MUSIC focuses on external Service Level Agreements (SLAs) and the fulfillment of those. Furthermore, it is not clear whether the proposed design is implemented and how planning will be realized.

### 8 Conclusions and future work

Pervasive computing has intrinsic heterogeneity of the involved devices where multiple communication protocols
can be used, which possess different QoS properties at runtime. This fact can be utilized to realize different QoS requirements at runtime through the self-adaption of communication protocols, where different protocols for different channels can be configured in a global optimized manner using GAs and/or CAAs. As a key part of self-adaption of protocols, we design and implement these protocols as reusable software connectors which can be changed dynamically at runtime. Other parts of the middleware then help to make these changes transparent to the end user, but also addressable through new web service endpoints.

From our tests, we can see that the overhead of protocol switching is quite acceptable in a pervasive web service computing environment. We have also evaluated the usage of three GAs, namely NSGA-II, MOCell, and FastPGA. As consistent with our initial evaluations, NSGA-II is still the best recommendation for the usage of self-management problem optimization.

In the future, we will test the self-adaption of protocols for the mobility in the pervasive systems where service end points suddenly disappear and come back again. We will investigate the usage of distributed GAs and combinatorial auctions for their potentials to be used to solve our problem. Currently only synchronized request/reply are supported, and only Bluetooth, Wired/Wireless TCP/UDP are supported, in the future, we will add the support for asynchronous calls, and add the support for other protocols. Also the QoS values used (for latency, power consumption etc.) are static, and they may change at runtime, we will taken this issue into account in the future.

Acknowledgements

The research reported in this paper has been supported by the Hydra EU project (IST-2005-034891). Thanks to Amro Al-Akkad for his comments on an early draft of the paper. The scenario in this paper is partially triggered in a discussion with Mads Ingstrup.

References


