pREST: A REST-based Protocol for Pervasive Systems

Witold Drytkiewicz Ilja Radusch Stefan Arbanowski
Radu Popescu-Zeletin
Fraunhofer Institute for Open Communication Systems
Technische Universität Berlin

{drytkiewicz,radusch,arbanowski,zeletin}@fokus.fraunhofer.de

Abstract

The convergence of embedded systems and wireless communication enables interconnection of electronic devices to render control and provide information to the user. Offices, apartments, and public spaces are or in near future will be able to deliver information and services to their occupants ranging from instant Internet access to configuration and control in a context dependent, personalized way.

Despite progressing internetworking and sophistication, we are still dealing with islands of functionality rather than the invisible computer envisioned by Mark Weiser. We believe that the spread and acceptance of smart environments will depend on common standards as well as a simple and flexible way to access data and devices and compose services from existing ones. A good example of such a system is the World Wide Web, whose success is mainly due to the simplicity with which all kinds of content can be published and referenced.

We present an access protocol to bring the Web's simplicity and holistic view on data and services to pervasive systems. Our approach is based on the Representational State Transfer architectural style and emphasizes abstraction of data and services as resources, interoperation via self describing data and service orchestration with loosely typed components. A particular concern is to provide for functionality in the absence of proxy nodes or infrastructure services like directory servers.

1. Introduction

Consider a simple security system, consisting of a sensor system, a digital camera, and a computer with Internet connection. The task to be accomplished is: each time someone triggers a (motion) sensor, the camera should take a picture and transmit it to the computer to publish it on the web.

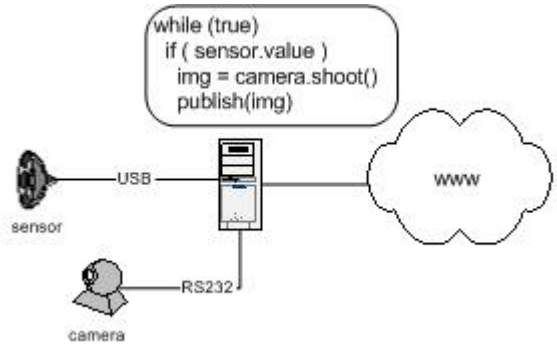

Figure 1. Logic, communication and user interaction often reside in the same code

Of all these components only the desktop computer is capable of running a middleware, say Java RMI. The camera and the photo sensor are equipped with simple embedded 8bit processors, capable of transmitting data via serial line or USB. Possibly they support TCP/IP [6]. But their processing power definitely isn't sufficient to make their data and services available as CORBA objects or via Java RMI.

To implement this scenario, the user would run a process to monitor the sensor and trigger the camera, whenever the motion sensor signals activity. The software could support a number of different cameras and sensors via device drivers or wrapper classes. Still, most of the logic and assumptions about the application scenario remains hidden within the code.

This configuration illustrates several restrictions when interconnecting heterogenous systems, especially if part of the service is based on Internet communication. First of all,

0-7803-8815-1/04/$20.00 ©2004 IEEE

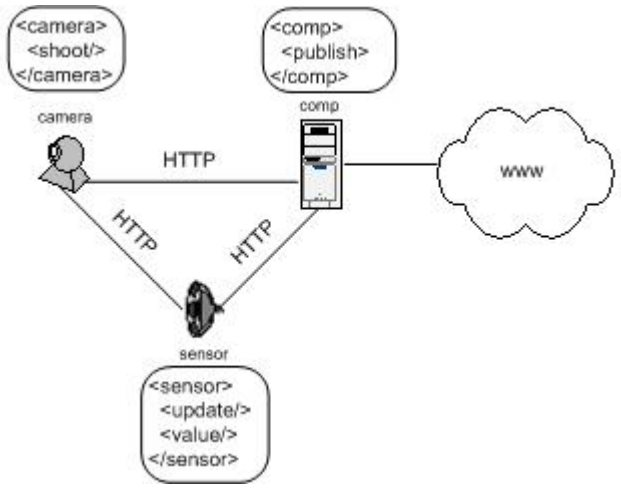

Figure 2. *pREST* provides functionality through interconnection of independent components

most middleware platforms require processing power and memory lying above, what simple sensors or actuators can provide. Commonly logic, communication, and user interaction is placed on the desktop machine, while the other nodes behave like peripherals. Secondly changes to the system configuration, like insertion of intermediate nodes or exchange of components are only possible as far as the software developer has forseen it - more precisely, as the interfaces allow it. Finally, user interaction for control and configuration of the individual parts is realized in a proprietary way, making it necessary to install additional software for each deployment.

As an alternative to these platforms we present pico-REST (*pREST*) an access protocol for pervasive systems based on the Representational State Transfer (REST) architectural style [7]. REST is a model for network applications which has proven successful on the World Wide Web. It emphasizes abstraction of data and services as resources and postulates a small set of commands with fixed semantics to access and manipulate resources. Compliant architectures are scalable, extensible, and allow independent deployment of components as well as *anarchic evolution* of the system, meaning that domain specific extensions such as content types can be introduced by consensus.

The same scenario based on the *pREST* protocol, consists of independent components exposing the data they produce and consume and the services they provide as independent communication endpoints referenced by URLs. The application logic resides mainly in the system configuration, i.e. in the interconnection of data producers, consumers, and service logic. Through the usage of human readable messages, runtime interconnection of services and data is eased.

Our implementation of the system is based on ASCII representation of data, XML description of components, HTTP transactions, and addressing with URLs. The resulting system is lightweight and network centric, provides built-in user interaction, and accounts for asymmetrical distribution of computational resources among participating nodes and loose coupling of components.

Network centrism Existing middleware platforms are library based, rather than network centric. Functions such as lookup, marshalling, and invocations are provided via a library, or auto-generated code. Invocations by components lacking a dedicated middleware layer are hard to perform, if at all. UPnP is an exception as it is based on open network protocols, rather than programmatical interfaces (still, UPnP requires components to process XML documents and support SOAP access).

While a library-based API ensures type conformance, correct invocation and an overall convenient programming model, it also creates a great deal of complexity, is less portable in a heterogeneous network, and results in genericity being preferred over performance [7].

A network based API on the other hand is an on-the-wire syntax with defined semantics for application interactions. It does not place any restrictions on the application code aside from the need to read/write to the network, but does place restrictions on the set of semantics that can be effectively communicated across the interface. On the plus side, it allows a custom implementation even on very constrained nodes, since the protocol handling code can be tailored to the functionality of the device itself.

Finally, it supports the usage of intermediary nodes which can inspect the transmitted messages and transform, redirect, or answer them, to provide for caching, in-network aggregation, and security.

User interaction The primary purpose of pervasive systems is to augment user activities in a possibly unobtrusive way. However, all of the previously mentioned middleware platforms lack built-in support for user interaction via existing clients. With the exception of UPnP allowing an optional user interface to be provided by devices, all middleware systems require custom applications to be run on clients for configuration and control. It is unlikely, that users will be willing to install custom software for each place they encounter. User interaction should therefore be integrated in the middleware for pervasive computing systems.

pREST is based on HTTP messages, thus compliant components can be accessed via telnet or Web clients, including micro browsers on smart phones and PDAs. Telnet access requires a basic understanding of the protocol by the user, and a lot of typing, but offers all features of *pREST* . Web access provides a more convenient way to invoke services with complex signatures, and parameters. However, *pREST* specific methods need to be mapped onto Web forms.

Asymmetric resources Existing middleware platforms require a relatively heavy-weight runtime, preventing their application on limited devices. Though lightweight implementations of these technologies are available [20][13][25], and extensions for spontaneous networking exist [21][17], memory requirements still lie in the range of tens of kilobytes. To deal with this limitation, the presented systems either introduce some concept of gateways translating requests to a proprietary protocol for limited devices, or silently assume such proxies being available.

However, relying on proxy nodes is insufficient for a number of applications and reduces scalability and reliability to partial failures. Though proxies increase efficiency and can provide advanced services, a basic level of functionality must be preserved even in the absence of intermediary nodes. The basic functionality any *pREST* node must provide is to get and set properties and invoke functions in response to plain text messages. This accounts for the basic client-server configuration.

Advanced services, such as storage of context, semantical information, and content adaptation and advanced querying, can be implemented on the application level, masked as resource access.

Loose coupling Common approaches to component interconnection require an agreement on interfaces published in form of an IDL or WSDL document. The creation of a compatible components involves the generation of stub objects to be linked against the application. This prevents a number of coding errors, such as wrong parameter types or mistyped identifiers already at compile time, but effectively prevents interoperation between independently developed components.

In pervasive systems, few components will be developed concurrently with a common interface definition to rely on. On the other hand, semantics that need to be communicated across the network are rarely more complex than 'an event occurred', 'retrieve property' or 'set value'.

Interfaces of components intended for interconnection should be equally simple allowing the delivery of plain values, such as numbers, strings or boolean values between communication endpoints defined at runtime. For more complex data to be exchanged, self describing messages such as XForms can be passed around between nodes, capable of processing them, as proposed in [1].

Since the correct and reasonable invocation of services on the server side is not accounted for in the application code, the user needs to be aided in the configuration of existing components, while the components need to verify the validity of user created configurations. This in turn requires a typing mechanism and semantic description at the interconnection and presentation level respectively.

2. Related Work

Systems aiming at integration of limited devices and spontaneous networking are Jini [21], UPnP [23] and Gaia OS [16] to name a few. Out of these UPnP has the most in common with our approach, i.e. the application of HTTP and XML for invocation and interface representation as well as an integrated mechanism for user interaction. In contrast to *pREST*, UPnP requires all participating nodes to understand XML and to support SOAP and GENA on top of HTTP and HTTP-UDP.

The integration of heterogeneous devices via HTTP has been investigated in a number of research projects. HP's Cooltown deals with the support for context management [4], localized services [1], and spontaneous interconnection of components [10] within pervasive systems using web technologies. IBM's BlueSpace project [2] investigates the integration of electronic appliances in workspace environments. The implementation is based on XML technology for representation of events, entities, and data.

Ueno et al. [22] present a system to control appliances via HTTP. Their architecture includes a web server which maps HTTP requests onto X.10 commands and provides an advanced lookup service, based on attribute queries. The AutoHAN system [19] integrates electronic appliances with the world wide web. Devices interact via asynchronous XML encoded messages. Both of these approaches utilize a centralized server or gateway node.

The application of XML component descriptors to generate user interfaces and as a programming layer for configuration and prototyping has been proposed by Hodes et al. [8]. The usage of XML descriptors to render device specific interfaces is presented and discussed by Roman et al. [18] and Ponnekanti et al. [15].

3. REST and *pREST*

pREST is based on the REST architectural style defined by Fielding [7] as a reference model for applications on the the World Wide Web. REST specifies a client-stateless-server architecture with implied use of caches. These constraints ensure that the final architecture is scalable, portable, and uses bandwidth efficiently. Furthermore advanced services such as advanced querying and name resolution can be externalized.

Services and data are abstracted as resources. REST requires all resources to support a small set of methods with fixed semantics, the most important being the retrieval of a resource representation via `GET`. A representation obtained this way contains the legal operations executable in this state, such as nested resources or parameters to be submitted to this service. This ensures an independent deployment of components and 'anarchic' evolution of the system. Further methods are `PUT`, `POST`, `DELETE` etc.

Instead of cache components, which are suitable for static content, *pREST* defines more general *intermediate nodes*, to provide services atop of simple control and configuration. While control and configuration can be achieved in a simple client-server manner with a generic client, services content based addressing, transformation, and context information can be provided transparently by advanced nodes. These services need to be masked as `GET` and `PUT` operations on URL-referenced resources to preserve transparency.

Besides constraints on interfaces and roles, REST defines an interaction pattern based on the exchange of resource identifiers and representations. In the request, the method and identifier of the target resource is submitted along with optional parameters. The server responds with a representation of the resource's state. This representation contains legal operations on that resource, which can be used for further requests. Server operation can then be viewed as state transitions, caused by user requests. The current state is presented to the user via a standard client - hence the name.

An example of that principle is browsing the web. A resource is requested by clicking on a link, and answered by transmitting the content of the requested resource to the client. This document usually contains further links to other resources. In *pREST* this pattern is applied to implement interface discovery. A request to a resource offering several services or kinds of data or expecting additional input is answered with a descriptor containing the various choices. It is then up to the user to interpret that notification correctly.

As an extension to the standard interaction pattern imposed by REST, which favors self descriptive XML representation we allow the delivery of plain text representations for primitive data types, to interconnect resource constrained devices. Secondly, to provide a lightweight data delivery mechanism and one-to-many communication, *pREST* allows the transmission of HTTP messages via UDP (HTTPU). Datagram communication avoids expensive connection establishment, but also facilitates an asynchronous communication pattern suitable for event driven applications. We believe HTTPU communication to be suitable for delivery and in-network aggregation of redundant data, such as sensor readings.

4. Resources

The primary mean of abstraction within *pREST* is a resource. Virtually anything that can be uniquely addressed with a URL is a resource: devices, services, and configurations as well as documents, pictures, and raw data. Resources can be nested, so that services and properties of a component being associated with URL are resources themselves. In this paper we let media and proprietary resource types aside and concentrate on resources represented by XML and plain text, since these are more likely to be exchanged between applications.

Fielding [7] emphasizes the equivalence between resource representations and identifiers, influenced by the synergistic relationship between URLs and internet resources on the web. We interpret a resource primarily as a uniquely identifiable, typed communication end point, capable of producing and/or consuming data, which reflects the dynamic aspect of interactions in pervasive systems.

While a client should perceive anything as a resource, some resources are more different than others. As far as distributed objects are concerned, three kinds of resources must be distinguished: complex, primitive, and services.

Complex resources are those containing at least one nested resource. and are represented by XML documents. These documents have no predefined tags, but rather contain the identifiers of nested resources as elements. The URL of a nested resource is constructed by appending its identifier to the parent reference.

Besides nested resources, descriptors may contain caption text to be displayed in a GUI generated from the document. Context and semantical information can be included in the descriptors using the Resource Description Framework (RDF) [12], with suitable ontologies specified in referenced OWL documents [24].

The descriptors of complex resources are static. This way even nodes unable to dynamically create XML documents, can provide a description of their functionality. The flexibility of XML documents allows the generation of user interfaces with XSL and enhancement of descriptors with context or semantic data,

without invalidation. Referencing data with URLs allows to store parts of the descriptors on external nodes.

Primitive resources differ from complex ones by having no nested elements, and are represented by plain ASCII strings. In contrast to complex resources, a `GET` operation on a primitive does not return a static document, but rather the value of this element. They represent a component's controllable and monitorable properties. Simple resources might also represent data objects, whose decomposition is not desirable or not sensible, e.g. images.

Primitive resources are also potential candidates for interconnection of components. In this case properties serve as sources or sinks of data. In the initial scenario the boolean data produced by the sensor value might be regularly transmitted to the trigger of the camera. The image output of the camera can be sent to the server, for instant publication on the web.

Services are a hybrid between primitive and complex resources. On the one hand, the invocation of a service returns a result value, which is analogue to a primitive, on the other hand the signature needs to be communicated in order for a human user (or an intelligent application) to invoke the service correctly. This is achieved via XML descriptors, similar to complex resources. The descriptor documents contain required parameters, exceptions, and a human readable description of the provided service. This information can be used both for dynamic invocation as well as for user interface generation.

A GET request for a service URL, is considered an interface request. The server responds with an XML representation of the signature containing the parameters and exceptions of this service. POST operations are interpreted as invocations of the service, with the parameters enclosed in the message body.

To provide additional information about resources, *pREST* supports a loose typing system via a HTTP headers and a MIME-like naming scheme similar to the one proposed in [1]. The typing of resources is realized via the headers `Accept` and `Provide`. In general, resources capable of consuming logical data, i.e. properties and services must be able to process `text/plain` messages. Complex resources may specify to provide `text/xml` for complex data types or media types such as `img/gif`.

A resource capable of consuming data of some kind, e.g. displaying images, is typed as `Accept: img/*`, and can be connected with resources providing image data, e.G. `Provide: img/gif`. More commonly the data exchanged between components will be logical values, such as numbers, strings, or boolean values. As a basis for property description we have chosen a small set of primitive types defined by XML schema. The types used by *pREST* consist of `int`, `boolean`, `string`, and `float`.

4.1. Example

Figure 3 shows an exemplary descriptor for a sensor node containing several kinds of nested resources. The particular type of the resource is at this stage visible only from the plain text description in the elements.

```
<sensor desc="Simple binary sensor">
   <context href="http://contextserver/sensor">
      Context information
   </context>
   <value href="./value">
      Current reading (true/false)
   </value>
</sensor>
```

Figure 3. A simple resource descriptor

The `context` element may contain information about the position, ownership, or relations to other resources of this node. Obviously this is a candidate for a complex resource. However, the sensor node might not be capable to store arbitrarily structured data. To overcome this limitation, the `context` element contains a reference to a remote resource stored on dedicated context server.

The `value` element is a primitive resource, representing the current reading of the sensor. A request for `http://sensor/value` returns the current value in the body of the response message.

5. Invocation

To enable independent deployment of components and interaction of components without knowledge of interfaces at compile time, *pREST* postulates a uniform interface for resources. Rather than a rich set of component specific methods, functionality arises from the combination of just a few methods, applicable to all resources. Differentiation of services is accomplished through addressing and combination of resources. This pattern also applied in SQL databases, where arbitrary complex applications can be constructed with `select`, `update`, `insert`, and `delete` combined with chains and filters.

5.1. HTTP methods

pREST uses the basic set of HTTP operations to manipulate resources. This allows *pREST* devices to be controlled via Web browsers or telnet. In addition, components

do not need to process XML documents (as opposed to SOAP and XMLRPC). Messages are plain ASCII strings, with fixed delimiters separating method, component identifier and parameters. Therefore invocation messages can be constructed even on limited devices.

The semantics that need to be communicated within pervasive systems are fairly simple. Property access, mapped on HTTP `GET` and `PUT` suffices for simple applications to query and configure devices. Invocation of services is expressed via `POST`, with the parameters passed as part of the URL.

In addition to these methods, *pREST* utilizes the HTTP `HEAD` method. `HEAD`, retrieves the header of a resource without the content, which can be used to determine the data accepted and provided a particular this resource. An exemplary session on our *pREST* enabled camera is presented in figure 4.

```
camera> GET /trigger HTTP/1.1

HTTP/1.1 200 OK
Content-type:  text/xml
Accept:  text/plain
Provide:  text/plain

<trigger>
  <parameters>
    <value type='boolean'/>
  </parameters>
  <exceptions/>
</trigger>

camera> POST /trigger HTTP/1.1
Content-type:  text/plain

true

HTTP/1.1 200 OK
Content-type:  text/plain
Accept:  text/plain

camera> HEAD /img HTTP/1.1

HTTP/1.1 200 OK
Content-type:  img/gif
Provide:  img/gif
```

Figure 4. Service invocation vs. interface request

The first request returns a descriptor of the `trigger` service. This is a rather simple service, taking a boolean value as a parameter and throwing no exceptions. The image data taken by the camera is stored under `/img` and is of type `img/gif` as a `HEAD` request on the respective property reveals. The `/img` resource can in turn be linked to the `/publish` service of the web server.

5.2. Component interconnection

Services in a *pREST* based system are realized through interconnection of independent of components. The data produced by one node is used to configure another or as a trigger for services. Interconnected devices form data paths, commonly starting with raw sensor readings being aggregated locally and transformed to high level events along their way to actuators.

In the exemplary scenario the sensor value acts as a trigger for the camera service, whereas the data produced by the camera is used as input for the web server. In a real world example, the readings would come from a network of sensors to reduce the probability of a false decision. The readings would be propagated hop by hop to the data sink and aggregated along the way. A consensus mechanism is simple enough, to be implemented in the sensor firmware.

The mechanisms to interconnect components in our *pREST* implementation are located at the application level. Consumers are interested in notifications upon property changes, caused by environmental changes, in case of sensors, or by application logic, in case of the camera. Likewise, the delivery varies from regular notifications to asynchronous delivery upon property change. For this reason, the policies of message generation and delivery should not be imposed by the interconnection protocol.

Sensors publish their properties specifying whether their values can be monitored, i.e. delivered to sinks. The producer-consumer relationships are created invoking a `subscribe` service on the sensor, expecting the property to be monitored, a URL where to send values to and a flag to indicate regular transmission or delivery upon property change along with the expiration time of the subscription to be submitted.

The subscribe service does not need to be invoked by the data consumer, as this would require the consumer to know the invocation syntax of a specific producer. Rather the descriptor of the `subscribe` service can be requested via a browser, and transformed into a Web form at the client side, using XSL transformations. The user can manually create a data link between participating nodes by completing a web form.

5.3. Parameters and return values

pREST resources can produce and consume any kind of data including images, word documents, and proprietary byte streams. However, most services and primitive resources exposed by server components accept logical values such as numbers, strings, or boolean values, representable as ASCII strings.

While a `PUT` request message and the response to a `GET` commonly contain a single value as their message content, multiple parameters of service invocations need to be bound. They are transmitted in url-encoded-post format, i.e. `attribute=value` pairs separated by ampersands.

Even though strings and numbers are the most common parameters and return values, transmission of data objects, consisting of several primitive values is often necessary. Proprietary encoding of complex data objects as strings can not be prevented by *pREST* , but is considered bad style, since it underruns the goal to allow composition of services and devices driven by content, not syntax.

A better solution is to break up a complex data object into primitive components, and use attribute names allowing an unambiguous mapping to the particular fields of a complex parameter. If a service requires a constructed type as a parameter, the primitive parts are grouped by a common prefix. The parameter submitted to the `subscribe` service for instance, is of type `SubscriptionData` combining the primitive parts mentioned above. These separate string, integer and enumeration types are prefixed with `'sub.'`.

Unlike library based middleware, *pREST* does not enforce a service to be invoked with the correct number of parameters and legal values. After all, an invocation is just a text message. However, the server can communicate an invocation failure in that case. It is then up to the user or application to interpret that error correctly.

6. Implementation

We have implemented the *pREST* protocol on two platforms, varying in resources and programming language capabilities. The components interconnected in our prototype are wrapped in a generic interface, allowing configuration, monitoring and control of devices and services. The `subscribe` service mentioned above used to interconnect components is defined as part of the Monitoring interface.

The interconnected components were a sensor node, and a software component emulating a lamp controllable via a software interface. The Java implementation of the lamp component is part of an architecture for applications in smart environments developed at Fraunhofer FOKUS.

6.1 Sensor hardware

The sensor node integrated in our scenario is the Embedded Sensor Board developed within the the Scatterweb Project at FU Berlin. The hardware is based on a low-power microcontroller (the Texas Instruments MSP430) with 2k of RAM and 65k Flash ROM for firmware and application code. Communication is provided via serial line, and radio. The sensing capabilities include light, microphone, vibration, and motion sensor as well as a passive infrared sensor.

Due to resource constraints, particularly the limited memory, processing of HTTP requests and invocations to the application are highly integrated. The buffers are shared between the IP layer and application, and requests processed partially and stored in a memory efficient way. The implementation is not divided into a generic middleware and an application process.

The implementation of the pREST protocol is based on the the IP stack by Adam Dunkels [6]. At the physical level, internet access is realized via a SLIP connection to a desktop machine, and via a broadcast radio link. Thus in a multihop scenario one sensor node must act as a base station.

In the implementation operational memory was the scarcest resource. Table 1 shows the memory footprint of the various layers. The firmware, providing for basic functions such as radio and serial line communication, sensor reading and communication with the application code uses 500 bytes of memory.

	Firmware only	Firmware + IP	Firmware + IP + REST app
Memory	500 byte	1818 byte	2008 byte
Code	17096 byte	23504 byte	37096 byte

Table 1. Memory footprint and and code size of *pREST*

The TCP/IP implementation takes up an additional 1300 bytes, most of it being buffer space for incoming packets. The implementation uses two buffers of 552 (512 bytes for data and 40 Bytes for the TCP/IP header). The interrupt handler uses one buffer to store raw data arriving on the radio interface or serial line. The other is used by the IP layer and the application.

Atop of the pure IP processing, the *pREST* and application layer require about 200 bytes for application state, such as subscription parameters and partial requests. For efficiency reasons HTTP requests and parameters are parsed one line at a time, and stored until the user completes the request. Unprocessed request parts cannot be left in the IP buffer, since any incoming IP packet overwrites the contents of the old one. Limiting request messages to one IP packet

might bring a memory gain here, but would also prevent telnet access.

To test the performance of our implementation we have measured the time it takes for the sensor to process a single request via serial line. To separate the impact of the different processing steps, we have measured the time for a request for two pre-generated messages of different sizes and one for a dynamically created message. As a lower bound for the latency we also list the times for a ping request. Table 2 lists the results.

Request Response	Static 70 bytes 99 bytes	Static 71 490	Dynamic 91 bytes 155 bytes	Ping 100 bytes 100 bytes
Min	200	261	220	40
Max	281	381	381	50
Avg	272	348	304	40

Table 2. Response times for various requests in ms

The first column lists the response times for a simple HTTP request with almost no processing at the application layer (besides looking up the request URL and pasting the HTTP status line and headers). The response times includes waiting for the complete request and copying each received line into an application buffer.

The second column shows the delays when requesting a interface descriptor as pregenerated content. The overhead to copy the static content to the IP buffer and fill in the content length is roughly 70 ms.

The construction of dynamic content as depicted in the third column, in this case the message contained all available sensor values, takes only about 30 ms longer than a static message of comparable size. Compared to the ping request, which does not involve checksum calculation or connection establishment, the processing at the transport and application layer consumes 80-90% of the total time.

As mentioned above, the application becomes less complex and significant performance gains can be achieved, by requiring the requests to be delivered in one packet. Processing can take place directly in the IP buffer, without copying request parts into an application buffer. The relatively high delay for connection establishment can further be compensated, by using TCP for creation of data links, and UDP for data delivery. The usage of text messages with a fixed format allows even a proprietary protocol, optimized for sensor networks to be used for transport.

6.2 Desktop hardware

In a full featured Java environment, we have implemented a wrapper to support *pREST* access to arbitrary objects. Providing object introspection and not limited by memory or processor constraints, it places no requirements on the components to be published (besides actually having any public methods or fields to be accessed). Server objects can be published as *pREST* resources without code modification. This way, software components can thus be interconnected with sensor nodes or other devices transparently, data provided by embedded hardware can be used just like user input.

7. Conclusion and future work

The reasons for employing HTTP as an interaction protocol have been discussed in a number of related publications. Implementations are lightweight and accessible via web clients [19], HTTP has a low barrier to entry, is device, service, and language independence and content centric [10]. Finally, the affinity between pervasive computing systems and the World Wide Web makes an integration via the same access protocol sensible [4]. Also a number of approaches to provide HTTP access to distributed objects [9][11][14] has been proposed so far.

However, the benefits of HTTP as an access protocol have often been reduced to the traversal of firewalls and its wide availability.We believe, that HTTP semantics, typing with MIME types and URL addressing offer a natural way to interconnect services and data. Particularly, services can be constructed through configuration, rather than programming.

Through interconnection with data paths, a collection of devices can be turned into a networked system to provide higher level services with just a few operations.

7.1. Design trade-offs

Still there is a price to pay for simplicity and flexibility. URL-encoded parameters might be convenient for users to type in, but are inefficient with machine-to-machine interaction. On the other hand since the primary goal of pervasive systems is user interaction, it seems a fair trade off. In addition complex parameters and return values have to be split up into primitives or encoded proprietary.

Stateless communication can handle temporary link breaks, but adds the overhead of connection setup to each operation. For applications where low response times are required, some strategy should be employed to decide whether to keep connections open for subsequent calls.

7.2. Future work

Links allow simple interconnection of devices, if the user knows exactly, which data to link. We will investigate automatic discovery of compatible resources, to allow

a more natural configuration by connecting e.g. `pda/out` to `display/in`, without specifying the concrete encoding. An automatic selection of transforming links to connect otherwise incompatible resources is also desirable.

The linking of communication properties and services offers a rudimentary scripting support for *pREST* components. Allowing operations on invocation parameters and conditional invocations would surely improve flexibility. This eventually leads to a definition of a general scripting language for *pREST*. Providing advanced services, like group addressing and in network aggregation of data along data paths also poses some interesting research questions.

References

[1] J. Barton, T. Kindberg, H. Dai, N. B. Priyantha, and F. A. bin ali. Sensor-enhanced mobile web clients: an XForms approach. In *Proceedings of the twelfth international conference on World Wide Web*, pages 80–89. ACM Press, 2003.

[2] P. Chou, M. Gruteser, J. Lai, A. Levas, S. McFaddin, C. Pinhanez, M. Viveros, D. Wong, and S. Yoshihama. Bluespace: Creating a personalized and context-aware workspace, October 2001.

[3] J. Cohen. Generic Event Notification Architecture. Internet draft.

[4] P. Debaty, P. Goddi, and A. Vorbau. Integrating the physical world with the web to enable context-enhanced services. Technical report, Mobile and Media Systems Laboratory HP Laboratories Palo Alto, 2003.

[5] S. D. O. DSIG. Platform Independent Model and Platform Specific Model for Super Distributed Objects. http://www.omg.org, 2003.

[6] A. Dunkels. Full TCP/IP for 8-bit architectures, May 2003.

[7] R. Fielding. *Architectural Styles and the Design of Network-based Software Architetures*. PhD thesis, Univ. of California, Irvine, 2000.

[8] T. D. Hodes and R. H. Katz. Enabling 'smart spaces': Entity description and user interface generation for a heterogeneous component-based distributed system. Technical Report CSD-98-1008, University of California, Berkeley, 6, 1998.

[9] D. B. Ingham, M. C. Little, C. J. Caughey, and S. K. Shrivastava. W3objects: Bringing object oriented technology to the web. *In Proc. Fourth Intl WWW Conf., Boston, Mass.*, December 1995.

[10] T. Kindberg, J. Barton, J. Morgan, G. Becker, I. Bedner, d. Caswell, P. Debaty, G. Gopal, M. Frid, V. Krishnan, H. M. aand C. Pering, J. Schettino, B. Serra, and M. Spasojevic. People, places,things: Web presence for the real world. *MONET*, 7(5), October 2002.

[11] D. Larner. Migrating the web towards distributed objects. Technical report, Xerox Corporation, 1996.

[12] E. Miller. An introduction to the resource description framework.

[13] Object Managemet Group. Minimum CORBA, 1998.

[14] J.-M. G. Philippe Merle, Christophe Gransart. Corbaweb: A generic object navigator. Technical report, INRIA, 1996.

[15] S. R. Ponnekanti, B. Lee, A. Fox, P. Hanrahan, and T. Winograd. ICrafter: A service framework for ubiquitous computing environments. *Lecture Notes in Computer Science*, 2201:56ff, 2001.

[16] M. Roman and R. H. Campbell. A middleware-based application framework for active space applications. citeseer.nj.nec.com/568916.html.

[17] M. Roman, C. K. Hess, A. Ranganathan, P. Madhavarapu, B. Borthakur, P. Viswanathan, R. Cerqueira, R. H. Campbell, and M. D. Mickunas. GaiaOS: An infrastructure for active spaces.

[18] M. Romn, J. Beck, and A. Gefflaut. A device-independent representation for services.

[19] U. Saif, D. Gordon, and D. J. Greaves. Internet access to a home area network. *IEEE Internet Computing*, 5(1):54–63, 2001.

[20] Sun Microsystems. J2ME building blocks for mobile devices.

[21] Sun Microsystems. Jini techonology core platform specification. Java specifications, Sun Microsystems, Oct. 2000.

[22] D. Ueno, T. Nakajima, I. Satoh, and K. Soejima. *Web-Based Middleware for Home Entertainment*, volume 2550, pages 206 – 219. Springer-Verlag Heidelberg, January 2002.

[23] UPnP device architecture.

[24] W3C. OWL - web ontology language overview.

[25] kSOAP. http://ksoap.enhydra.org/.