Adaptive Preference Specifications for Application Sessions

Christine Julien

Mobile and Pervasive Computing Group
The Center for Excellence in Distributed Global Environments
The University of Texas at Austin
c.julien@mail.utexas.edu

Abstract. In ubiquitous computing applications, mobile participants must be empowered to opportunistically connect to services available in their local environments. Our previous work has elucidated a model for allowing applications to specify the functional properties of the services to which they need to connect. Our framework then connects applications to dynamic resources through the use of a novel suite of *application sessions*. In this paper, we revisit this framework to devise a mechanism for applications to specify preferences for one service provider over another. In this investigation, we argue that these preferences are actually provided by a set of session participants: the application itself, the service provider, and, more surprisingly, the network that connects the application and the provider. We develop a framework for each of these parties to specify preferences among various allowable connections. We demonstrate not only what kinds of properties can be expressed in our framework but also implementation paths for integrating them into the communication and application support infrastructure.

1 Introduction

In ubiquitous computing, software and hardware resources are available embedded in a user's environment. The service concept provides an intuitive abstraction through which applications can gain access to remote resources. In dynamic ubiquitous computing environments such as aware homes [1] or first responder situations [2], applications opportunistically connect to a set of locally available resources that change due to the application's (or user's) mobility. Such environments commonly rely on *mobile ad hoc networks* to provide network connectivity. In mobile ad hoc networks, devices are disconnected from any wired infrastructure and instead communicate directly with one another using wireless radio signals. Such networks employ multihop routing protocols that use intermediate devices as routers for communicating partners that are not directly connected.

Application sessions [3] enable applications to select resources from the immediate environment based on their functional properties. The approach is simple in that it uses non-deterministic selection to connect the service requester to *any* available resource that matches the request. However, it only accounts for the static properties that define the capabilities of a particular resource; it does

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 78–89, 2006.

not allow applications to express the fact that a resource with particular non-functional properties is preferable to another resource.

In this paper, we create an expressive preference function that we incorporate into the application sessions framework. The function allows all parties that participate in the service interaction to express their non-functional requirements regarding a particular service binding. This includes the resource user, the resource provider, and the devices that support the connection between the provider and user. Characteristics that are likely to have an impact on the selection of a particular resource include: the relative mobility of the user and the resource, the proximity of the user to the resource, the reliability of a resource, the battery power of the devices involved, etc.

This work differs from previous work because connections and the preferences associated with them are determined dynamically in a changing environment. In addition, all policy evaluations must be accomplished in a distributed and ad hoc fashion because no infrastructure exists to facilitate service selection. The specific novel contributions of this work fall in two categories. First, we define a framework for preference specification. Second, we provide implementation paths for incorporating these preferences into the application sessions framework.

This paper is organized as follows. First we overview the original application sessions model. We then extend the model to include preference functions. Section 4 describes related work, and Section 5 concludes.

2 The Application Sessions Model

The application sessions model [3] defined a set of interactions between ubiquitous computing applications and services available in the applications' immediate surroundings. Our model explicitly separates the user program (i.e, the *application*) from the session management infrastructure that manages coordination with available providers. The only knowledge shared between the two are a specification (*spec*) that describes the desired service, and a provider handle (p) that allows the application to access the provider the infrastructure connects it to.

Services and requests are described using semi-structured data [4], an approach common among description languages [5] and tuple based systems [6]. We use eLights [7], a flexible, lightweight tuple space implementation. Each device maintains a local *tuple space* where it stores information in *tuples*. Service providers describe service properties using tuples; a location service that provides readings once a second may be described as:

$$\langle (\mathit{service}, \text{location}), (\mathit{frequency}, 1\text{ sec}) \rangle$$

A service description may include additional tuple fields, e.g., format of information, error rate, etc. Service requests are encoded as templates (or patterns) over the description tuples. Content-based matching determines whether a description tuple matches a request template. An example template is:

$$\langle (\mathit{service}, = \text{location}), (\mathit{frequency}, < 30\text{ sec}) \rangle$$

This request matches location services that have a frequency of less than 30 seconds. A communication protocol underlying the application session framework delivers request templates to providers [8], where matches are evaluated against tuples in the provider's local tuple space. No intermediate lookup service aids in this process; providers respond directly to requests they receive. This autonomy afforded by mobile ad hoc networks allows the framework to apply to dynamic ubiquitous computing environments.

We focus on three specific session types from our model: the *query session*, *provider session*, and *type session*. Detailed application examples that motivate each session type can be found in [3]; they are omitted here for brevity. Each session is represented as an assignment to a local handle that the requestor subsequently uses as a proxy for the discovered service. Throughout our description, the *entails* ($\models$) relation indicates that a resource satisfies a specification, i.e., in $p \models spec$, service p satisfies *spec*. The selection of a matching provider uses *non-deterministic assignment* [9] to indicate that a provider is selected from any that satisfy the specification. A statement $x := x'.Q$ assigns to x a value x' nondeterministically selected from among the values satisfying the predicate Q. If an assignment is not possible, the statement aborts; we assume this results in assigning ϵ (a null value) to x.

A *query session* is a simple, one-time request for data from some remote service. The application should be connected to a single matching service for the duration of this interaction. The session provides no long-lived interaction with the selected provider. We write the semantics of a query session as:

$$\boxed{p = spec} \triangleq p = p'.(p' \models spec \wedge p'.\texttt{reachable})$$

The expression in the box denotes the particular session semantic. In this case, the query semantic is expressed by assigning the specification to the handle p. The value assigned is nondeterministically selected from all services that satisfy the specification and are reachable. The reachable relationship models the requirement that the two devices can communicate with each other, perhaps using a multihop path in the ad hoc network.

The *provider session* supports applications that connect to a remote service and perform several operations with that specific provider. This is useful, for example, when an interaction produces state at both endpoints that is necessary for subsequent interactions. The operational semantics can be written as:

$$\boxed{p \leftarrow spec} \triangleq p = p'.(p' \models spec \wedge p'.\texttt{reachable})$$
if $p \neq \epsilon$ **then**
$\langle$**await** $\neg p.\texttt{reachable} \rightarrow p = \epsilon\rangle$[1]
fi

[1] The $\langle$**await** $B \rightarrow S\rangle$ construct [10] allows a program to delay execution until the condition B holds. When B is true, the statements in S are executed in order. The angle brackets enclosing the construct indicate that the statement is executed atomically, i.e., no state internal to S is visible outside the execution of S.

In a provider session, the infrastructure maintains the connection to a particular resource given network dynamics. As long as the infrastructure can maintain a connection to the initial provider, the provider session is maintained. When the connection fails, the handle is assigned ϵ, which effectively notifies the application that the requested resource is no longer available.

In contrast to the previous sessions, the particular service provider supplying the resource in a *type session* can change during the session, as long as the new provider also satisfies the request specification. An example is a connection to a location server; the particular provider servicing a mobile device's requests for location readings is likely to change over time, but programming the application is simplified if this dynamic binding is transparent to the application. We express the type session formally as:

$$\boxed{p \Leftarrow spec}$$
$$\triangleq\ p = p'.(p' \models spec \wedge p'.\texttt{reachable})$$
while $p \neq \epsilon$ **do**
$\langle$**await** $\neg p.\texttt{reachable} \rightarrow p = p'.(p' \models spec \wedge p'.\texttt{reachable})\rangle$
od

If an attached provider becomes unreachable, the infrastructure attempts to locate a new provider that *is* reachable and matches the specification. As long as such a provider is available, the application remains connected to one, nondeterministically chosen from those that meet the requirements.

These session types do not completely address the needs of ubiquitous computing applications. What an application truly wants is the ability to request that it is connected to the best available provider for some measure of "best" (for example, the closest provider). We rectify this problem by introducing an expressive preference function as an extension to the existing framework that continues to hide the complexity of creating highly interactive ubiquitous applications.

3 Specifying Preference

The framework described above assumes that each provider that matches the functional specification is equally well suited. In this work, we introduce a function (f) that evaluates properties of a potential matching service and its hosting device, properties of the application and its hosting device, and properties of the network that connects the two. The latter is important because our framework supports dynamic connections between applications and services in mobile ad hoc networks where ordinary devices must serve as routers for communication among other hosts in the network. As such, the cost of supporting communication between two peers in the network (i.e., the application host and the service provider) has impact on other devices in a manner that is not commonly captured by end-to-end quality of service approaches such as [11,12].

Our general framework for defining the preference function f relies on three partial cost functions: $f_a(p)$, which defines the cost to application a of selecting

a particular provider p; $f_p(a)$, which defines the cost to provider p of servicing application a; and $f_n(a,p)$, which defines the cost of a network path between the devices hosting the application (a) and the potential provider (p). In combination, the overall "global" preference function can be defined as:

$$f(a,p) = \alpha f_a(p) + \rho f_p(a) + \nu f_n(a,p)$$

where α, ρ, and ν are system-specified constants that can place varying emphasis on the three different components under different operating conditions. The provider that best satisfies this function at any given instant has a cost of:

$$c_{optimal} = \langle \min a, p : p \models spec :: f(a,p) \rangle^2.$$

All of the partial cost functions ($f_a(p)$, $f_p(a)$, and $f_n(a,p)$) return values between 0 and 1, as described below, and $\alpha + \rho + \nu = 1$, so that the result of evaluating the global preference function for any application/provider pair is a value between 0 and 1. Each of the partial cost functions is also allowed to return ∞, which effectively vetoes that party's participation in the session.

In the remainder of this section, we first revisit the model to show the revised semantics of sessions with preferences. We then explore each of the partial cost functions in more detail, describing how its behavior is implemented within our tuple-based model. This is especially important in the case of network costs, where our approach allows $f_n(a,p)$ to be computed as part of the underlying communication protocol, thereby incurring minimal additional overhead.

3.1 Preference Based Application Sessions

Incorporating preferences into the application sessions model requires augmenting the operational semantics listed in Section 2 to ensure that the "best" provider is chosen based on the global cost function. In this section, we express the semantics in terms of the global cost function; in the subsequent sections we show how these global semantics are implemented using the partial cost functions described above. We abuse our own notation slightly here by writing $f(p)$ to indicate the global preference function $f(a,p)$ because these definitions are written from the application's perspective (i.e., a is fixed in all cases). If p is not reachable from a in the mobile ad hoc network, then the cost function has a value of ∞, that is, $\neg p.\texttt{reachable} \Rightarrow f(p) = \infty$. How this is implemented is described in Section 3.4.

The preference-aware query session simply selects the service provider to connect based on minimizing the preference function:

[2] In the *three-part notation*: $\langle$**op** *quantified_variables* : *range* :: *expression*$\rangle$, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three part expression is the identity element for **op**, e.g., *true* if **op** is $\forall$ or 0 when **op** is min.

$$\boxed{p = spec/f}$$
$$\triangleq \; p = p'.(p' \models spec \wedge p'.\texttt{reachable} \wedge \langle \forall \pi : \pi \models spec :: f(p') < f(\pi) \rangle)$$

The provider session is similar; the best provider is selected, and no subsequent reselection is performed:

$$\boxed{p \blacktriangleleft\!\!- spec/f}$$
$$\triangleq \; p = p'.(p' \models spec \wedge p'.\texttt{reachable} \wedge \langle \forall \pi : \pi \models spec :: f(p') < f(\pi) \rangle)$$
if $p \neq \epsilon$ **then**
$\langle$**await** $\neg p.\texttt{reachable} \rightarrow p = \epsilon \rangle$
fi

The preference-aware type session is more complicated because not only must it ensure that it initially selects the best match, but it has to constantly monitor the available providers to ensure that this match remains the best one available:

$$\boxed{p \Leftarrow spec/f}$$
$$\triangleq \; p = p'.(p' \models spec \wedge p'.\texttt{reachable} \wedge \langle \forall \pi : \pi \models spec :: f(p') < f(\pi) \rangle)$$
while $p \neq \epsilon$ **do**
$\langle$**await** $\neg p.\texttt{reachable} \vee \langle \exists \pi : \pi.\texttt{reachable} \wedge \pi \models spec \wedge f(\pi) < f(p) \rangle \rightarrow$
$p = p'.(p' \models spec \wedge p'.\texttt{reachable} \wedge \langle \forall \pi : \pi \models spec :: f(p') < f(\pi) \rangle)$
od

The statement inside the loop ensures that the application is reconnected if a better provider is available. As above, this assignment ultimately sets the handle to ϵ if no satisfactory provider is available.

3.2 Implementing Application Preferences

The most obvious of the three components of the preference function is the application requesting access to the service. Factors that influence the application include aspects such as:

- service fidelity (i.e., the error rate in responses from the provider)
- service availability (i.e., the percentage of time the provider is responsive)
- proximity of the provider (i.e., location-dependent interactions tend to favor closer providers)
- relative mobility (i.e., if both the application's host and the provider's host can move, stable connections will involve low *relative* mobility)

The portion of the global preference function related to application preferences is $\alpha f_a(p)$. Effectively, the application provides a function that takes as a parameter a provider (or more specifically, attributes of a provider) and generates a cost value. This is done for every potential provider, enabling the application to select the lowest cost provider. Service providers generate attribute tuples in their local tuple spaces that provide up-to-date information about the attributes described above. As an example, a provider may output a velocity tuple:

$$\langle (speed, \text{my\_speed}), (direction, \text{my\_direction}) \rangle$$

As the provider's velocity changes, it removes this tuple and inserts a new tuple representing the updated velocity. Service requests evaluating the cost for this provider can access this information asynchronously to generate values for the cost to this provider. The provider may not have a tuple for every attribute; if a client request requires information about an attribute but the provider does not provide it, the connection cannot be made.

The application's cost function is provided as an *active tuple* as was introduced in the Linda model [6]. An active tuple differs from the previously described *passive tuples* in that it can contain uncompleted computation. In our case, the active tuple encapsulates the computation that calculates the cost function's value for a particular provider:

$$\langle (source, \text{requester\_id}), (cost_a, \texttt{cost}_\texttt{a}(\ldots)) \rangle$$

where the *source* field's value indicates the unique id of the requesting device, and the $cost_a$ field's value actually calculates the cost for the application to use a particular provider. The communication protocol broadcasts this active tuple to every host reachable in the network. This is excessive, since the network could be very large; Section 3.4 shows how this broadcast is restricted to only provider devices that are in a reasonable range to service the request.

The code implementing $\texttt{cost}_a(\ldots)$ for our example based on the relative mobility between the service requester and the service provider has the form:

```
cost_a(my_velocity)
   rd( [spec] )
   v = rdp( ⟨(speed, ? ), (direction ? )⟩ )
   if(v == null)
      return ∞
   else
      return relative_velocity(v, my_velocity)
```

The first line is a blocking tuple space operation, a `rd`, which waits until it encounters a tuple matching its argument (in this case the template representing the service request). When the active tuple encounters such a match, the blocking `rd` operation returns, allowing the cost function to continue. At this point, the active tuple knows that it is at a location hosting a service provider that matches the request. The second line is a probing (non-blocking) `rdp` operation that looks for a tuple matching the provided template; in this case, a tuple containing speed and direction information (where the values for the fields are unrestricted, as indicated by the "?"). If no matching tuple exists, the operation returns `null`, and the cost cannot be computed. If a matching tuple does exist, the result is used to compute the relative velocity of the two devices (normalized to be between 0 and 1). This example uses only one attribute of the provider, but more attributes can be incorporated by looking for additional attribute tuples. When the function returns, it automatically replaces the function portion of the

active tuple (the second field) with the returned value, resulting in a passive tuple:

$$\langle (source, \text{requester\_id}), (cost_a, \texttt{relative\_velocity}) \rangle$$

The underlying communication protocol responds to the presence of this tuple and returns it to the requester. Each device within the network receives a copy of the original active tuple; each device that supports a matching service executes the active tuple, generating a cost for that service. Therefore, the requester receives a cost tuple for each potential provider, allowing the application to select the best option.

3.3 Implementing Provider Preferences

The second participant in the session that desires to to influence service selection is the service provider itself. Factors that might influence whether or not a provider wants to participate in a session include:

- current provider load
- current battery level of provider device
- announced intended length of usage by the application requesting access (or duration of session)
- periodicity of requests from the application

The portion of the global preference function related to the provider specified preferences is $\rho f_p(a)$. When a provider makes a service remotely available, it also specifies a preference function that takes as a parameter an application (or more specifically, attributes of a particular application and its request) and generates a cost value for servicing that request.

The implementation of the provider preference adds to the process elucidated in the previous section. Instead of the communication protocol responding to a two-field passive tuple (the tuple containing the requester's id and the application cost value), the protocol responds only to a three-field active tuple:

$$\langle (source, \text{requester\_id}), (cost_a, \text{cost\_value}), (cost_p, \texttt{cost}_\texttt{p}(\ldots)) \rangle$$

Before the communication protocol responds to this tuple, the provider device removes the two-field tuple generated by evaluating the application's cost function and replaces it with the above three-field tuple by inserting its own cost function. This cost function is partially evaluated with respect to having values filled in for needed provider attributes (e.g., current load), and when it arrives at the requester, it reads attributes about the application requesting the service (e.g., the frequency of requests). Once the communication protocol transports this three-field tuple back to the requester (using the tuple's first field, which uniquely identifies the requester), the provider's cost function completes its execution using tuples read from the requester's local tuple space.

3.4 Implementing Network Preferences

While the first two stakeholders (the application and the service provider) are obvious, a third, often overlooked component is the network. Ubiquitous computing applications like those mentioned in Section 1 are supported by mobile ad hoc networks in which the nodes themselves serve as routers for connections between devices that are not directly connected. These intermediate nodes have a vested interest in ensuring the connections selected ensure the longevity of the network as a whole. Factors that play into network preferences include:

- aggregate bandwidth available on potential transmission path
- number of network hops
- battery power available on intermediate nodes
- latency of the network connection

The portion of the global preference function related to the network is $\nu f_n(a, p)$. This function has a static definition applicable across the entire network and known to all applications, but the values that influence the cost calculated for a path are themselves dynamic. We have so far assumed a broadcast-based communication protocol in which every request is delivered to every other device in the network. Our communication protocol that incorporates the network cost function embodies additional intelligence. A mobile ad hoc network is made up of a set of devices connected by a graph in which vertices in the graph are wireless devices and edges in the graph are direct connections between devices that are physically close enough to be within communication range. A mobile ad hoc network routing protocol can dynamically impose a tree on this graph where the root of the tree is the requesting device and paths to other devices emanate out from the root. Our resource requests move along these paths, recalculating the network cost at every hop. If the network cost ever exceeds its allowable threshold (provided statically by the network deployer to each node), the message kills itself. Our network cost function allows more sophisticated definitions, though the simple definitions can still be used to halt propagation.

When using all three cost functions in conjunction, the process changes slightly again from the previous section. The static network cost function is provided to every device, and any requesting device must place this cost function in its request tuple:

$$\langle (source, \text{requester\_id}), (cost_n, \texttt{cost}_\texttt{n}(\texttt{cost}_\texttt{a}(\ldots), \ldots)) \rangle$$

where the network cost function ($\texttt{cost}_\texttt{n}(\ldots)$) is the active portion of the tuple, and the application's cost function is invoked when a match is encountered. The first line of the application's cost function defined above is no longer necessary; this matching of the application's specification against the provider's description is performed within the network cost function.

The implementation of the network cost function has the following structure:

```
cost_n(cost_a(...), ...)
   current_cost = 0
   max_net_cost = threshold
   while(true)
      if rdp( [spec]) != null
         out(⟨(source, requester_id), (cost_n, current_cost), (cost_a, cost_a(...))⟩
      update_cost(current_cost)
      if current_cost < max_net_cost
         forward_self
      else
         out([garbage collecting active tuple])
         return ∞
```

The `rdp` operation checks to see if the current device has a service that matches the application's request. If so, the function generates a dedicated tuple for responding from this provider and places it in the provider's local tuple space. Processing of this tuple proceeds as described above; the application's cost function is evaluated first. When it finishes, the provider removes the tuple and replaces it with a tuple containing its unevaluated cost function. The result tuple now contains four fields, including a value for the network cost function.

Whether the current provider matched or not, the network cost function continues by updating the network cost stored within the active tuple. The single statement `update_cost(current_cost)` encodes a more complicated process that may involve carrying some state from one node to another and/or reading values stored in local tuples (e.g., local available bandwidth information or remaining battery power). When the network cost function generates a cost value that exceeds a specified threshold, it performs a sequence of steps that ensure that the request no longer propagates and that it leaves no residue on the current provider. This process suffices completely for query and provider sessions; type sessions require the active tuple to remain resident and send updates back to the requester if any of the cost values change. This allows the requester to reconnect to a better provider as soon as one becomes available.

4 Related Work

Research projects have increasingly focused on providing applications dynamic access to a changing set of resources. We highlight the most relevant projects, especially with respect to how applications specify constraints or preferences on selected resources. Many projects have focused on mediating quality of service requirements by leveraging object mobility [13,14] to enhance application responsiveness and network-wide performance metrics. These approaches focus on bringing objects closer to clients instead of on the notion that the clients themselves are mobile and resource usage may be inherently location-dependent.

Network sensitive service selection [15] observed the differences between performing user-side resource selection (where the user collects necessary information about available providers) and provider-side resource selection (where a

provider collects information about potential users). Work founded on these observations introduced the ability for applications to include network parameters and requirements in resource requests.

In moving from network-sensitivity to awareness of quality of service (QoS), service efficiency has been defined as a tradeoff between service coverage and cost [16]. This work is extended in [17] which provides guaranteed availability of a multimedia service in dynamic ad hoc networks using a combination of algorithms that includes predicting network partitions. This work focuses on optimal creation and placement of service instances in dynamic ad hoc networks, and therefore is sufficient only for non-location-dependent software services. Our approach addresses the discovery of resources (both physical and software resources) that are available in a local environment.

Work more closely related to our approach [11] differentiates QoS parameters into metrics and policies and considers both constraints on the user of a resource and constraints on the provider of that resource. This work does not naturally accommodate dynamically changing QoS measurements and is limited to traditional performance-style measurements (like bandwidth, reliability, load, etc.). Our approach handles application-level requirements (e.g., location, mobility, etc.) and defines three categories of constraints (application-, provider-, and network-specific constraints) instead of just two. This allows us to consider the impact that a peer-to-peer interaction has on the rest of the network, not just its impact on the direct participants. Other work [18] introduces formal modeling tools that enable optimal service compositions to be selected given a static set of QoS requirements. Our approach focuses on the ability of an infrastructure to dynamically adapt such selections in response to changes in the underlying network and service infrastructure.

A final important component of the work that was described in this paper is its ambition to simplify the development of adaptive ubiquitous computing applications. Along that same vein, previous work has created middleware solutions to enable developers to easily specify the relationships between their applications and QoS metrics [12]. In a similar manner, DySOA [19] enables service compositions to dynamically evaluate the network status and adapt the system at runtime to maintain a set of specified QoS parameters.

5 Conclusions

Our approach explicitly separates preferences into three categories, allowing the application, the resource provider, and the network to each specify preferences with regard to a potential resource interaction. At runtime, these preferences are dynamically evaluated, and connections between applications and resource providers are automatically maintained to ensure that these preference functions are maximized and that no constraints are violated. This style of interaction is essential to applications which function in long-lived ubiquitous computing environments where applications' interactions are inherently location, environment, and task-dependent.

Acknowledgments

The author would like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment. This research was funded, in part, by the NSF, Grant # CNS-0620245. The views and conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. Kidd, C., Orr, R., Abowd, G., Atkeson, C., Essa, I., MacIntyre, B., Mynatt, E., Starner, T., Newstetter, W.: The aware home: A living laboratory for ubiquitous computing research. In: Proc. of CoBuild. (1999)
2. Malan, D., Fulford-Jones, T., Welsh, M., Moulton, S.: CodeBlue: An ad hoc sensor network infrastructure for emergency medical care. In: Proc. of BSN. (2004)
3. Julien, C., Stovall, D.: Enabling ubiquitous coordination using application sessions. In: Proc. of Coordination. (2006)
4. Abiteboul, S.: Querying semi-structured data. In: Proc. of ICDT. (1997) 1–18
5. Christensen, E., Gubera, F., Meredith, G., Weerawarana, S.: Web services description language (WSDL) 1.1 (2001) Current as of 2005.
6. Carriero, N., Gelernter, D.: Linda in context. Communications of the ACM **32**(4) (1989) 444–458
7. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proc. of FSE. (2002) 21–30
8. Julien, C., Venkataraman, M.: Resource-directed discovery and routing in mobile ad hoc networks. Technical Report TR-UTEDGE-2005-01, Univ. of Texas (2005)
9. Back, R., Sere, K.: Stepwise refinement of parallel algorithms. Science of Computer Prog. **13**(2-3) (1990) 133–180
10. Andrews, G.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison Wesley (1999)
11. Liu, J., Issarny, V.: QoS-aware service location in mobile ad hoc networks. In: Proc. of MDM. (2004) 224–235
12. Nahrstedt, K., Xu, D., Wichadakul, D., Li, B.: Qos-aware middleware for ubiquitous and heterogeneous environments. IEEE Comm. Magazine (2001) 140–148
13. Grimm, R., Davis, J., Lemar, E., MacBeth, A., Swanson, S., Anderson, T., Bershad, B., Borriello, G., Gribble, S., Wetherall, D.: System support for pervasive applications. ACM Trans. on Computer Systems **22**(4) (2004) 421–486
14. Holder, O., Ben-Shaul, I., Gazit, H.: Dynamic layout of distributed applications in FarGo. In: Proc. of ICSE. (1999) 163–173
15. Huang, A.C., Steenkiste, P.: Network-sensitive service discovery. Journal of Grid Comput. **1**(3) (2003) 309–326
16. Li, B.: QoS-aware adaptive services in mobile networks. In: Proc. of IWQoS. Volume 2092 of LNCS. (2001) 251–268
17. Li, B., Wang, K.: Nonstop: Continuous multimedia streaming in wireless ad hoc networks with node mobility. IEEE Journal on Selected Areas in Comm. **21**(10) (2003) 1627–1641
18. Yu, T., Lin, K.J.: Service selection algorithms for composing complex services with multiple QoS constraints. In: Proc. of ICSOC. (2005) 130–143
19. Siljee, J., Bosloper, I., Nijhuis, J., Hammer, D.: DySOA: Making service systems self-adaptive. In: Proc. of ICSOC. (2005) 255–268