Performance Model Building of Pervasive Computing*

Andrea D'Ambrogio and Giuseppe Iazeolla
Dept. of Computer Science S&P
University of Roma TorVergata
E-Mail: {dambro,iazeolla}@info.uniroma2.it

Abstract

Performance model building is essential to predict the ability of an application to satisfy given levels of performance or to support the search for viable alternatives.

Using automated methods of model building is becoming of increasing interest to software developers who have neither the skills nor the time to do it manually.

This is particularly relevant in pervasive computing, where the large number of software and hardware components requires models of so large a size that using traditional manual methods of model building would be error prone and time consuming.

This paper deals with an automated method to build performance models of pervasive computing applications, which require the integration of multiple technologies, including software layers, hardware platforms and wired/wireless networks. The considered performance models are of extended queueing network (EQN) type. The method is based on a procedure that receives as input the UML model of the application to yield as output the complete EQN model, which can then be evaluated by use of any evaluation tool.

1. Introduction

Pervasive computing is based on the use of heterogeneous platforms consisting of various computing devices that collect and elaborate data for use by a multiplicity of fixed or mobile users. The platforms also run distributed and heterogeneous software to both coordinate hardware devices and provide services to customers. Such a complex of components is performance critical both from the user point of view and from the service provider point of view. It is thus important to introduce appropriate methodologies to build performance models and evaluate such models. Since of the large number of software and hardware components, the model building activity should be based on innovative automated methods rather than on traditional manual methods, based on experience and intuition, that would be error-prone and time consuming.

This paper introduces an automated method to build performance models of pervasive computing architectures. Circumstances in which it is important to make use of automated model building methods in pervasive computing are, e.g., when it is necessary to:

- predict the performance of a new pervasive application at development time;
- re-design or re-configure the software application in order to maintain a given level of service;
- evaluate the applications scalability so that additional users may be accommodated without impact on the existing ones;
- identify the components (software, hardware or network elements) that are critical to reaching a satisfying end-to-end performance.

So far, the model building methods that have been proposed roughly differ for [2, 10]:

- the type of performance model that is produced (e.g., queueing network, petri net, markov chain, process algebra, etc.);
- the type of software models from which the performance model is produced (e.g., UML models, execution graphs, architecture description languages, etc.);
- the automation degree, or the extent to which a method implementation can be easily produced to automatically build the model.

* This work was supported by the Italian Ministry of Education, Universities and Research (MIUR), in the framework of the FIRB-Perf project, by the University of Roma TorVergata Research Program on "Performance Modeling of Service-oriented Architectures for Quality Verification" and by the University of Roma TorVergata CERTIA Research Center.

This paper focuses on performance models of *queueing network* (QN) type. The proposed method receives as input the UML model of the pervasive software application and yields as output the QN model, ready to be evaluated by use of any evaluation tool.

Model building methods proposed in literature are partly manual in nature and based on experience and intuition [4, 16] and partly semi-automatic [3, 5]. In order to qualify model building as automatic, the process should be procedure-driven according to the following main steps:

a) translate the UML model into an intermediate software model based on the *execution graph* (EG) formalism [17, 18];

b) derive the queueing network graph, consisting of service centers and connections between centers, based on the platform configuration;

c) parametrize the model in b) by use of the model in a), in other words identify the centers' service rates and the routing probabilities among centers.

The obtained QN can then be evaluated to yield performance predictions in terms of response time, throughput, utilization, etc.

This paper introduces an automated method to build a QN model from UML models. The obtained QN is an *extended queueing network* (*EQN*), or a QN that can model resource simultaneous possession, fork-join software primitives, split primitives, etc. It is assumed that step a) has already taken place, in other words an EG has been derived from a set of UML documents (namely, the use case diagram and the set of sequence diagrams that realize the use cases), as described in details in [3, 6].

The method receives as input a part of the UML model (i.e., the deployment diagram) and the EG of step a), and yields as output the complete EQN model, which can then be evaluated by use of any evaluation tool.

Being procedure-driven, the method can easily be implemented into tools that enable application designers to easily introduce performance prediction activities as an integral part of their work. In particular, the method can be implemented into an XMI-based tool, that receives as input two XMI documents (i.e., the execution graph and the deployment diagram) and produces the XMI document of the resulting EQN model, as fully described in [8].

It is worth noting that one of the problems that arises in model building is taking into consideration the various layers of software between the system platform and the user application. The model should then be produced according to a multi-level approach to explicitly consider all abstraction levels of the system [9]. Such an aspect however is out of the scope of this paper, which concentrates instead on the automation of the model building method and will thus only deals with a single level of abstraction (i.e., the user application level).

Paper content is divided into Section 2, which describes the pervasive computing architecture, and Section 3, which illustrates the proposed method to build the corresponding EQN model.

2. The Pervasive Computing Architecture

The goal of pervasive computing is to enable users to access data or run software applications from any site in any situation ("*anywhere and anytime*" [19]). To achieve this universal utility, any pervasive software application should be continuously available even in presence of user or terminal mobility, should transparently adapt to the current context of the user and reflect the individual preferences of the user.

Figure 1 illustrates an example architecture that can support various kinds of services (e.g., providing personalized services about weather conditions, traffic, public utilities, security, etc., to fixed and mobile users). As shown in Figure 1, the architecture integrates multiple technologies, such as sensor networks, wired and wireless networks, hardware servers, fixed and mobile user terminals.

Sensor networks are composed of a large number of sensor nodes, which are densely deployed within a geographic area to monitor a wide variety of environment conditions, such as temperature, humidity, vehicular movement and the presence or absence of certain kinds of objects along with current characteristics such as speed, direction, and size of an object [1]. A sensor node is made up of a sensing unit (sensor plus analog to digital converter or digital sensor), a processing unit (associated to a small storage unit), a transceiver unit and a power unit. The analog signals produced by the sensors are converted to digital signals and then fed into the processing unit that manages the procedures that make the sensor node carry out the assigned sensing tasks. The transceiver unit connects the node to the sensor network.

The various *sensor networks* are connected to the back-end Internet network by means of *proxy servers*, which process, store and relay the events gathered by the sensors on a given site (there may exist more than one such sites - *local site 1* through *local site n* - as in Figure 1)[11]. Such events traverse the sensor network (a low-rate and low-power wireless network, e.g., IEEE 802.15.4) to reach the proxy server that acts as a front-end for the sensor network.

On the left side of Figure 1, users interact with the application to get personalized information. Two type of users are considered, the *fixed user* and the *mobile user*.

The fixes user uses a standard web browser to execute a client application that, upon successful authentication, sends a request about the status of a given site. The request

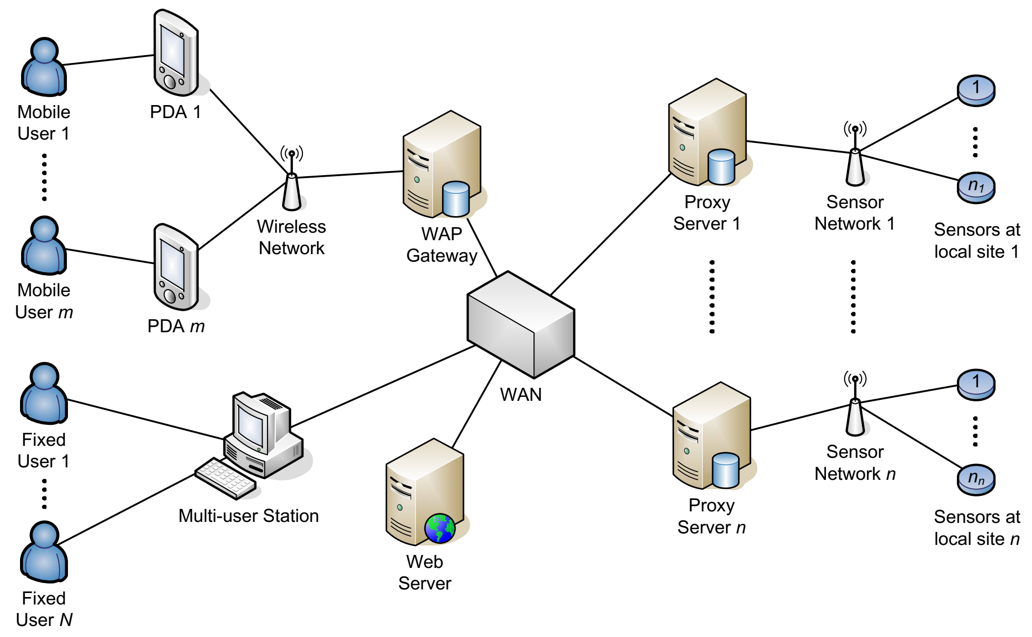

Figure 1. Pervasive computing architecture

is sent through the *WAN* (a wide area network of DSL type) to the web server, where the application server first retrieves the user preferences and then forwards the request through the WAN to the relevant proxy server.

The mobile user uses a *WAP* (*Wireless Application Protocol*) browser to execute a *user agent*, i.e., a program running on the *PDA* (*Personal Digital Assistant*) that acts on the users behalf. The user agent sends a WAP request to the *WAP gateway* through a wireless network (e.g., a GSM network with GPRS - General Packet Radio Service). The type of information requested by the user depends on the current location of the user, as well as on the user preferences and the PDA capabilities. To this purpose, the PDA location and capabilities are stored in the user agent profile (e.g., a CC/PP profile [12]) sent with the WAP request, while the user preferences are stored at the web server site. The WAP gateway authenticates the user, decodes the WAP request and forwards it as a HTTP request to the web server, through the WAN. The web server first retrieves the user preferences and then queries the relevant proxy server. The response obtained from the proxy server is then sent back to the WAP gateway that finally encodes it, for appropriate visualization on the PDA screen, before transmitting the response to the user's PDA [13].

It is assumed that the pervasive application is described by a software model of UML type [14]. The model consists of a *deployment diagram* and a set of a *execution graphs*[1], described in Section 2.1 and Section 2.2, respectively.

2.1. The Pervasive Computing Deployment Diagram

The *deployment diagram* (*DD*) describes the pervasive computing platform and devices, with the allocation of software components onto platform devices.

Figure 2 illustrates the considered DD, which consists of nodes that represent:

- a *Multi-user Station*, which executes the `ClientApp` component and includes a CPU (`MScpu`), a disk (`MSdisk`) and a terminal (`MSterm`) operated by fixed users;
- number *m PDAs* (`PDA1` through `PDA`*m*), which execute the `UserAgent`*i* component and include a CPU (`PDA`*i*`cpu`), a storage (`PDA`*i*`store`) and a terminal (`PDA`*i*`term`) operated by mobile users;
- a *WAP Gateway*, which executes the `WapServer` component and includes a CPU (`WGcpu`) and a disk (`WGdisk`);
- a *Web Server*, which executes the `AppServer` component and includes a CPU (`WScpu`) and a disk (`WSdisk`);

1 The execution graph is not part of the UML notation, but results from the merge of a set of UML diagrams (namely, the use case diagram and the set of sequence diagrams that realize the use cases) from which it can be automatically derived, as fully described in [3] and [7].

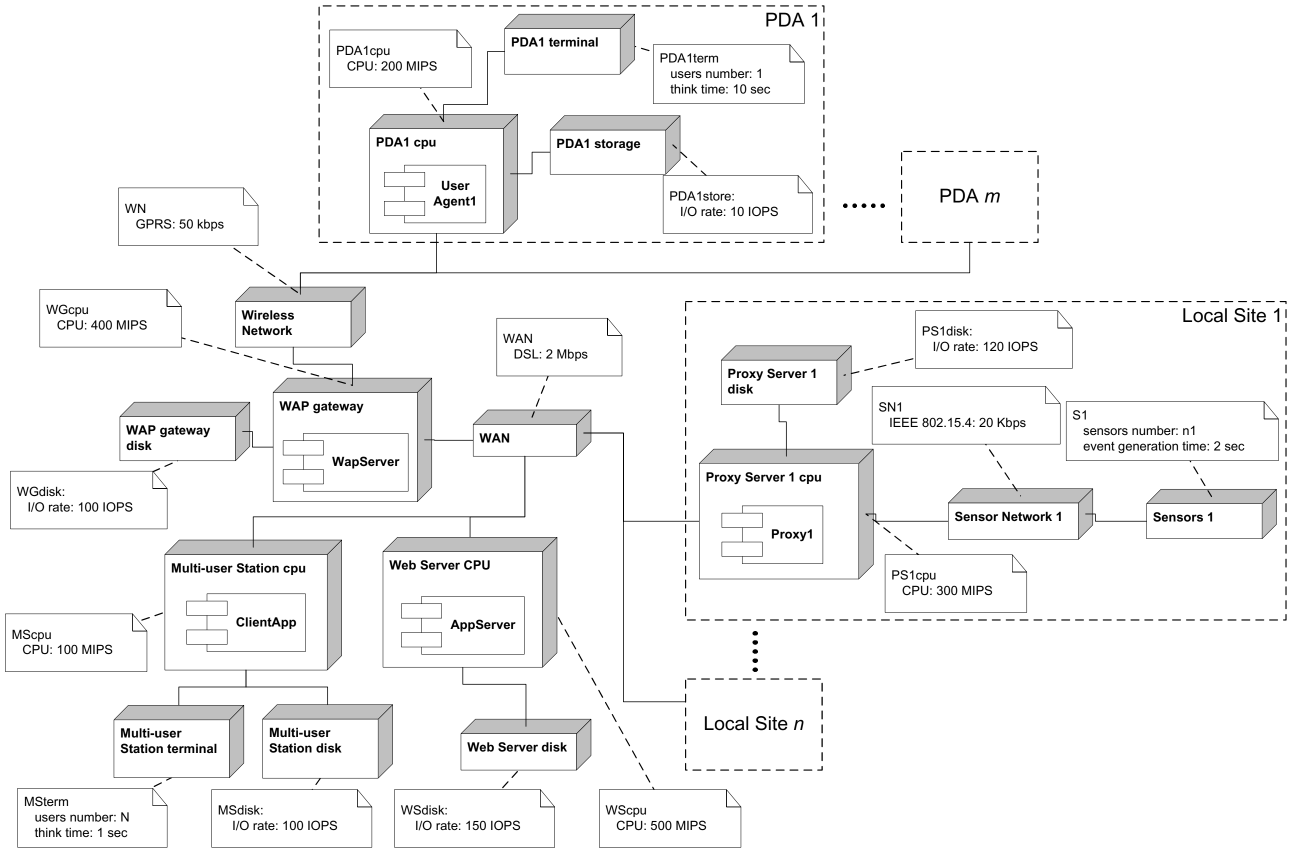

Figure 2. The pervasive computing Deployment Diagram (DD)

- number n *Proxy Servers* (PS1 through PSn), which execute Proxy$i$ components and include a CPU (PS$i$cpu) and a disk (PS$i$disk);
- a *wide-area network* (WAN), that connects the Multi-user Station, the WAP Gateway, the Web Server and the Proxy Servers;
- number n *Sensor Networks* (SN1 through SNn), that connect sensor nodes at the n local sites;
- number n sets of *Sensors* (S1 through Sn), each one consisting of a number of sensor nodes (n_1 through n_n) that produce monitoring events at a given local site;

Notes associated to DD nodes give the performance-oriented characteristics of each node, in term of capacities (for nodes of *cpu* and *disk* type), bandwidth (for nodes of *network* type), number of users with think time (for nodes of *terminal* type) and number of sensors with event generation time (for nodes of *sensor* type). For the sake of clarity, such annotations are expressed in natural language rather than by use of the stereotypes and the tag values of the UML Performance Profile [15].

The performance critical components can be identified into:

- each proxy server PSi ($i = 1..n$), loaded by a large number n_i of sensors and by the interactions with the web server;
- the WAP gateway, loaded by number m PDAs;
- the web server WS, loaded by the WAP gateway and number N fixed users;
- each PDAi ($i = 1..m$), that can be loaded by a complex software application even though individually used;
- the multi-user station, loaded by number N fixed users.

Building the performance model may thus imply deriving a set of n submodels for the proxy servers, m submodels for the PDAs in addition to the WAP gateway submodel, the web server submodel and the multi-user station submodel.

Further submodels are the n sensor networks submodels, the wireless network submodel and the WAN submodel. For the sake of brevity, such submodels are not detailed in this paper and are each represented by a single DD node, as shown in Figure 2. In case of performance criticality they can however be adequately studied in any detail, as described in [5].

2.2. The Pervasive Computing Execution Graph

According to conventions, an *execution graph* (*EG*) is a standard flow graph enriched with resource usage data, specified by associating a so-called *resource demand vector* to each EG block [17]. An EG block denotes a piece of straight-line code or a sequence of program statements that are all executed. A resource demand vector D_i, associated to block i, consists of M components (where M is the number of DD nodes), each component relating to a distinct node in the DD (e.g., a CPU node, a DISK node, a WAN node, etc.):

$$D_i = (D_{i,0}, ..., D_{i,k}, ..., D_{i,M-1})$$

The generic $D_{i,k}$ gives the number of elementary operations (e.g., n_{CPU}, n_{DISK}, n_{WAN}, etc.) the EG block i demands to DD node k.

Figure 3 illustrates the EGs for the considered pervasive application. The three EGs in the upper part of Figure 3 represent, from left to right, the typical execution flow of fixed users, mobile users and sensors (at local sites), respectively.

The EG of a fixed user starts with a call for user's authentication (`call authenticate()` block). The authentication is performed by a *software server* block (`authenticate()` block)[2]. After user' authentication the EG enters a CASE block and proceeds by calling, with probability p_i, the `getStatus(site`*i*`)` ($i = 1..n$) operation, which is performed again by a software server block. Finally, the EG visualizes the resulting page (`displayPage()` block).

The EG of the mobile user i ($i = 1..n$) first retrieves the user profile (`getProfile()` block) and than sends a WAP request (`WapRequest()` block). The request is processed by the `processRequest()` block, whose response is finally visualized on the PDA screen of the mobile user (`displayResults()` block).

The EG of a sensor of any site (*site 1* through *site n*) simply consists of a request to record the event produced by a sensor (`call proxy`*i*`.recordEvent()` block) followed by its execution performed by the `recordEvent()` software server block.

The EGs in the lower part of Figure 3 give the details of software server blocks, with side bars, in the upper part. Each sub-EG is easily identified by the corresponding labels on BEGIN and END blocks.

2 Software servers are blocks that may be accessed by a more than one user simultaneously. The maximum number of simultaneous users is specified by the degree of multiplicity on the left side diamond [3]. A block with side bars is called *extended block*. The presence of side bars indicates that the software server is itself a sub-EG consisting of several blocks that are specified elsewhere.

Figure 3 also gives demand vectors of each EG block. The demand vector elements are specified according to the vector format illustrated at the bottom of each EG or sub-EG. As an example, the execution of the `call authenticate()` block in the fixed user's EG requires 1 access to `MSterm`, 20×10^3 instructions of `MScpu`, 2 accesses to `MSdisk`, 0 instructions of `WScpu`, 0 accesses to `WSdisk` and 1 access to `WAN`.

3. The EQN Model Building Procedure

This section illustrates the model building procedure. The procedure receives as input the EGs and the DD of the pervasive application and yields as output the EQN model, and consists of 6 steps:

step 1) introduce a service center of the EQN for each node of the DD;

step 2) build the *centers reachability graph* (*CRG*) (in other words, the graph that specifies interconnection between centers) on the basis of the DD communication links;

step 3) transform the CRG into a primordial EQN, that only includes waiting queues in front of service centers (where necessary);

step 4) introduce EQN job classes, one class for each EG;

step 5) by use of the EGs, transform the primordial into a more complete EQN, that includes job classes, direction of links between centers and additional centers (fork, join, split, lock, free, etc.), where necessary;

step 6) complete the EQN by specifying the service rate of each service center and the routing probabilities between centers, for each job class.

At step 1 of the procedure, each node of the DD is transformed into a service center of EQN. The EQN will thus include one *MScpu* center, one *MSdisk* center, one *WAN* center, one sub-graph for each local site and for each PDA, and so forth.

Step 2 of the procedure builds the CRG by adding undirected links between the centers identified at step 1. An undirected link is added for each DD communication link, as illustrated in Figure 4.

At step 3, the model building procedure proceeds by introducing waiting queues (with FCFS service discipline) in front of all centers of CPU or DISK type. The WAN, the wireless network and the sensor networks are modeled by aggregate delay centers, according to assumptions, while the sensors, the multi-user station terminal and the PDA terminals are modeled by infinite service centers, as by conventions.

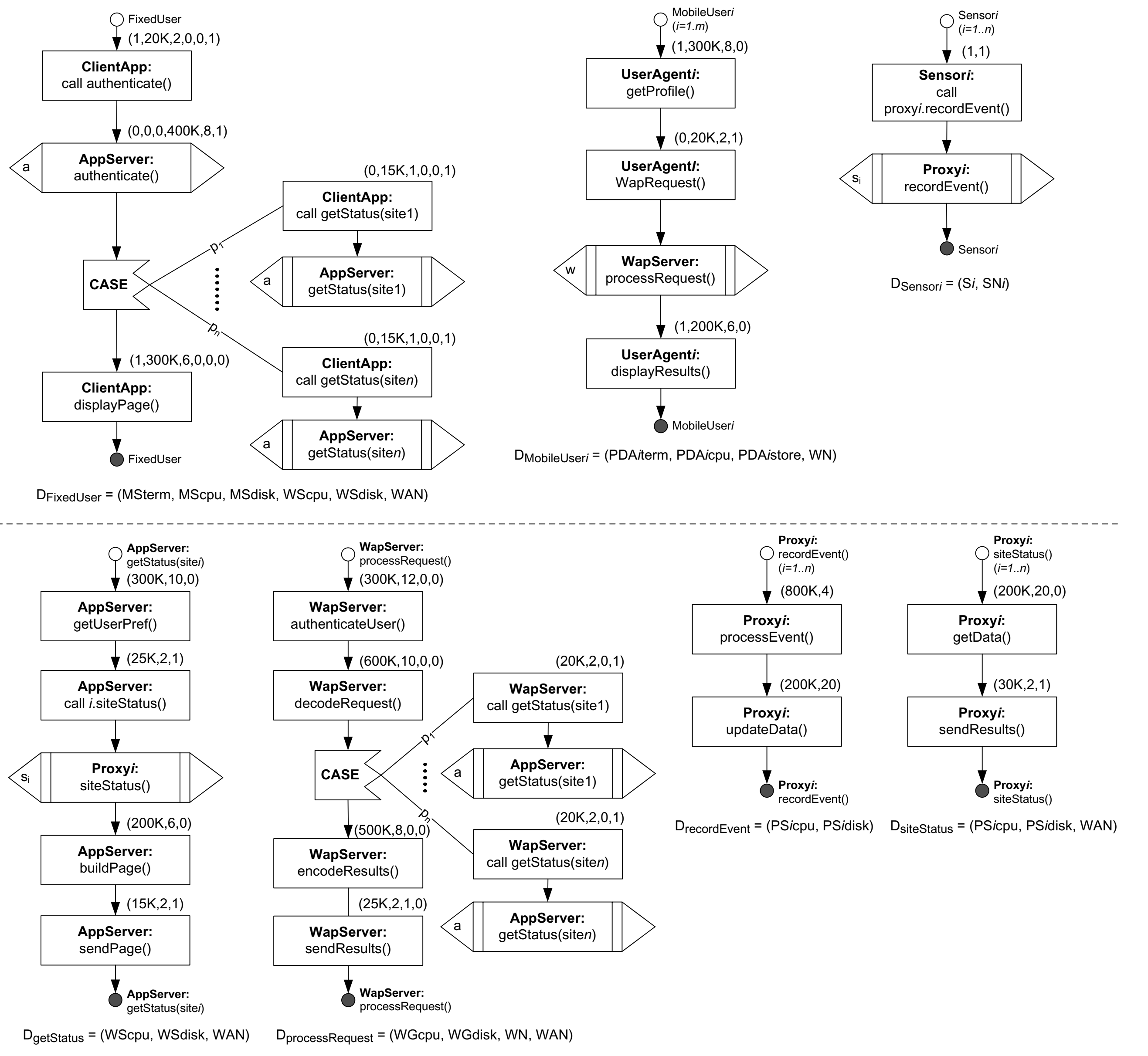
FixedUser
(1,20K,2,0,0,1)
ClientApp: call authenticate()
(0,0,0,400K,8,1)
a
AppServer: authenticate()
CASE
p_1
p_n
(0,15K,1,0,0,1)
ClientApp: call getStatus(site1)
a
AppServer: getStatus(site1)
(0,15K,1,0,0,1)
ClientApp: call getStatus(siten)
a
AppServer: getStatus(siten)
(1,300K,6,0,0,0)
ClientApp: displayPage()
FixedUser
$D_{FixedUser}$ = (MSterm, MScpu, MSdisk, WScpu, WSdisk, WAN)
MobileUseri (i=1.m)
(1,300K,8,0)
UserAgenti: getProfile()
(0,20K,2,1)
UserAgenti: WapRequest()
w
WapServer: processRequest()
(1,200K,6,0)
UserAgenti: displayResults()
MobileUseri
$D_{MobileUser i}$ = (PDAiterm, PDAicpu, PDAistore, WN)
Sensori (i=1..n)
(1,1)
Sensori: call proxyi.recordEvent()
s_i
Proxyi: recordEvent()
Sensori
$D_{Sensor i}$ = (Si, SNi)
AppServer: getStatus(sitei)
(300K,10,0)
AppServer: getUserPref()
(25K,2,1)
AppServer: call i.siteStatus()
s_i
Proxyi: siteStatus()
(200K,6,0)
AppServer: buildPage()
(15K,2,1)
AppServer: sendPage()
AppServer: getStatus(sitei)
$D_{getStatus}$ = (WScpu, WSdisk, WAN)
WapServer: processRequest()
(300K,12,0,0)
WapServer: authenticateUser()
(600K,10,0,0)
WapServer: decodeRequest()
CASE
p_1
p_n
(20K,2,0,1)
WapServer: call getStatus(site1)
a
AppServer: getStatus(site1)
(20K,2,0,1)
WapServer: call getStatus(siten)
a
AppServer: getStatus(siten)
(500K,8,0,0)
WapServer: encodeResults()
(25K,2,1,0)
WapServer: sendResults()
WapServer: processRequest()
$D_{processRequest}$ = (WGcpu, WGdisk, WN, WAN)
Proxyi: recordEvent() (i=1..n)
(800K,4)
Proxyi: processEvent()
(200K,20)
Proxyi: updateData()
Proxyi: recordEvent()
$D_{recordEvent}$ = (PSicpu, PSidisk)
Proxyi: siteStatus() (i=1..n)
(200K,20,0)
Proxyi: getData()
(30K,2,1)
Proxyi: sendResults()
Proxyi: siteStatus()
$D_{siteStatus}$ = (PSicpu, PSidisk, WAN)

Figure 3. The Execution Graph (EG) of the considered pervasive application

At step 4, job classes are introduced in the EQN, one class for each EG. According to Figure 3, number $1+m+n$ classes of jobs are introduced, namely:

- 1 fixed user's job class (*FixedUser*), associated to the corresponding EG in the upper-left part of Figure 3;
- number m mobile user's job classes (*MobileUser_1* through *MobileUser_m*), associated to the m replicas of the EG in the upper-medium part of Figure 3;
- number n sensor's job classes (*Sensor_1* through *Sensor_n*), associated to the n replicas of the EG in the upper-right part of Figure 3, with *i=2* (local site1);

Step 5 of the procedure is carried out by visiting the EGs corresponding to job classes introduced at step 4 and incrementally building the EQN. For each EG, the function `build_EQN` is executed, which takes as input the EG and the CRG and returns the EQN, as described in Section 3.1.

Step 6 of the procedure is finally carried out to obtain the EQN parameters for each job class. To this purpose, the function `parameterize_EQN` is executed, which takes as input the EQN built at step 5 and returns the parameterized EQN, as described in Section 3.2.

The so obtained complete EQN can then be evaluated by use of any evaluation tool (e.g., QNAP, RESQ, etc.) to yield

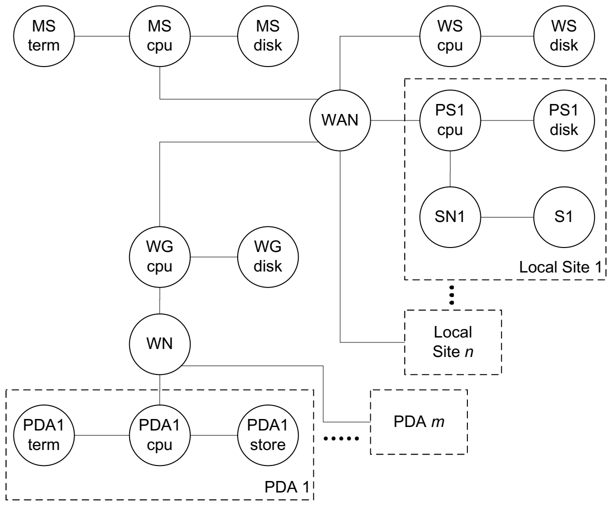

Figure 4. Centers reachability graph (CRG) obtained from the DD in Figure 2

the required performance indices of the considered pervasive application, such as response times, throughputs and utilizations[3].

3.1. The function `build_EQN`

The function `build_EQN` takes as input the following four parameters:

1. `EG`: the considered EG (or any its sub-EG);
2. `EGocc`: a `double` that specifies the number of occurrences of the EG specified in the previous parameter (i.e., the number of times it is executed);
3. `CRG`: the CRG;
4. `startC`: the starting EQN center visited.

and returns the EQN.

An example assignment of such parameters and execution of the function `build_EQN` for the “FixedUser” EG in Figure 3 is as follows:

```
EG = the ``FixedUser'' EG;
EGocc = 1;
CRG = the CRG obtained at step 1;
startC = the reference center (i.e., ``MSterm'')
EQN = build_EQN(EG,EGocc,CRG,startC);
```

The rationale of the function `build_EQN` is that the succession of the EG blocks with their demand vectors and the succession of the demand vector elements within each EG block provide the necessary information to determine the flow of EQN jobs onto the links given by the CRG built at step 2 of the method, thus giving directions to such links [5].

The function is driven by a variable that identifies the EG block currently visited and by a variable that identifies the EQN center currently visited by the corresponding job. In the course of its execution, the function updates the entries `(h,k)` of a matrix `A(jc)` that collects data about the number of times the job of class `jc` moves from center `h` to center `k`, being `jc` the variable that identifies the job class corresponding to the EG.

When visiting a given block of the EG, depending on the block type, an execution of `build_EQN` yields a new EQN portion and adds this to the EQN already obtained from the previously visited EG blocks. This is illustrated in Figure 5 that shows the addition of the sub-EQN resulting from the application of `build_EQN` to the software server `authenticate()` of the *FixedUser* EG in Figure 3. The `Token Pool` with number *a* tokens and the related `LOCK` and `FREE` centers are introduced. According to the CRG and to what specified by the demand vector of the `authenticate()` block, the job flows from the last visited center (i.e., `WAN`), then enters the LOCK center to acquire a token and finally cycles between the `WScpu` and the `WSdisk` to execute the 400K `WScpu` instructions into 8 visits to `WSdisk`. After the last visit to the `WSdisk` the job returns to the `WScpu` (ninth visit), from where it enters the `FREE` to release the token and finally return to the `WAN`. The corresponding update of matrix `A` by use of demand vector data proceeds as follows:

$$
\begin{aligned}
A_{WAN,LOCK} &= A_{WAN,LOCK} + 1\\
A_{LOCK,WScpu} &= A_{LOCK,WScpu} + 1\\
A_{WScpu,WSdisk} &= A_{WScpu,WSdisk} + 9\\
A_{WSdisk,WScpu} &= A_{WSdisk,WScpu} + 8\\
A_{WScpu,FREE} &= A_{WScpu,FREE} + 1\\
A_{FREE,WAN} &= A_{FREE,WAN} + 1
\end{aligned}
$$

It is easily seen that the 400×10^3 CPU instructions specified by the demand vector are executed by 9 visits to the `WScpu`.

In a similar way, all remaining sub-EQNs are derived to obtain the complete EQN, illustrated in Figure 6, that will be reached when the final block of the EG has been visited.

3.2. The function `parameterize_EQN`

The function `parameterize_EQN` takes as input the EQN and the matrix `A` built by the `build_EQN` function and returns the EQN parameters for each job class. The entries of matrix A are used to obtain both the service rate μ of each service center and routing probabilities between centers, for each job class.

3 EQN evaluation results are not presented here, the focus of the paper being limited to model building.

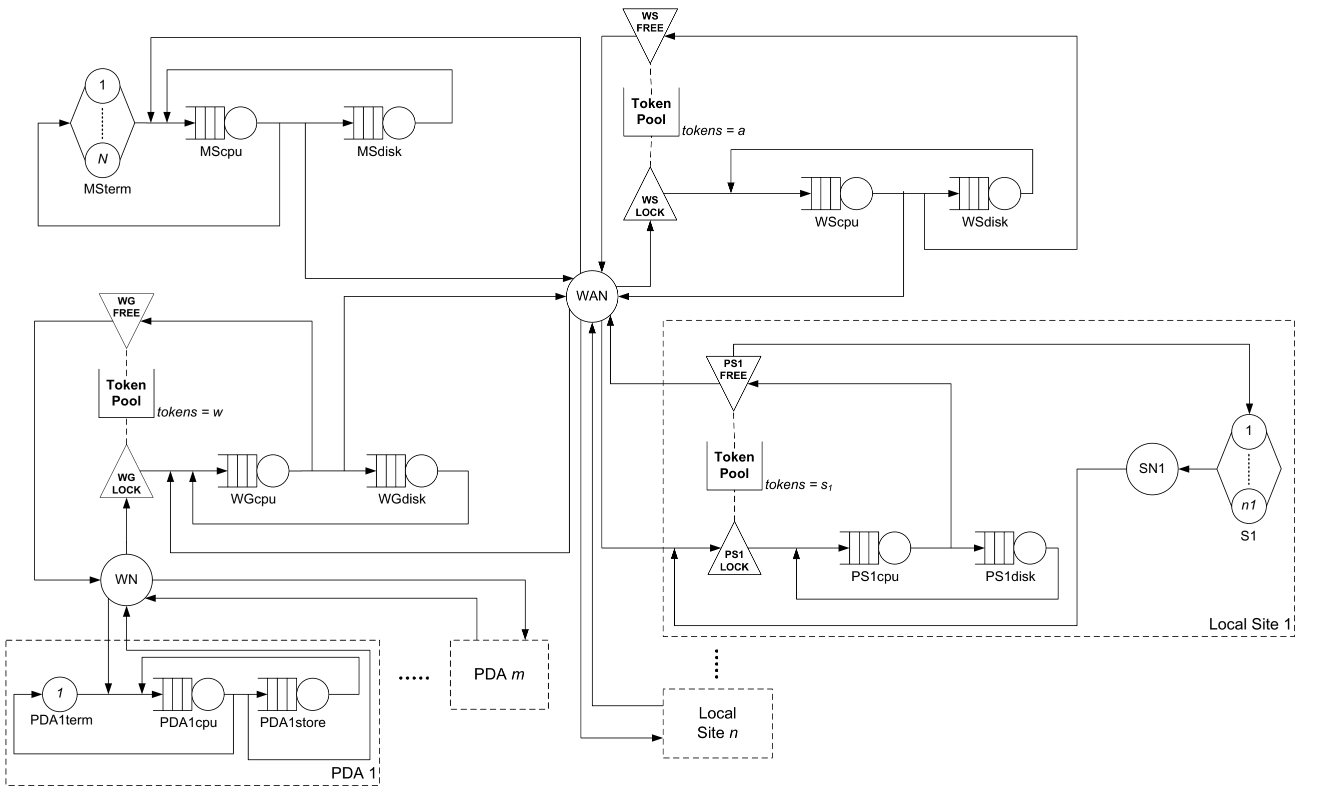

Figure 6. EQN built from the EG in Figure 3 and the DD in Figure 2

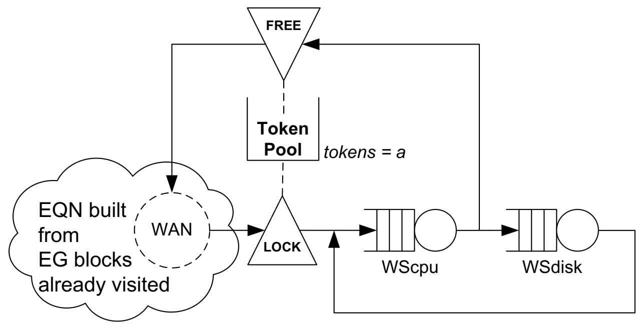

Figure 5. Sub-EQN resulting from the application of the function `build_EQN` to the software server `authenticate()` in Figure 3

To this purpose, let's denote by $\mu_k(jc)$ the *mean service rate* of center k for jobs of class jc, and by $P_{k,h}(jc)$ the *probability* to move from center k to center h for jobs of class jc jobs. Then:

$$\mu_k(jc) = \frac{V_k(jc)}{ops_k} \times capacity_k \qquad \text{(if } k \text{ is not a CPU)}$$
$$\mu_k(jc) = \frac{V_k(jc)}{instr_k(jc)} \times capacity_k \qquad \text{(if } k \text{ is a CPU)}$$

where, for each center k of the CRG:

- $V_k(jc)$ denotes the number of visits to center k by jobs of class jc and is obtainable from matrix A (see [5]);
- $P_{k,h}(jc)$ is also obtainable from matrix A (see [5]);
- $instr_k(jc)$ denotes the number of instructions the class jc jobs demand to center k, being k a center of CPU type;
- ops_k denotes the mean number of elementary operations requested for each visit to center k, being k a center of type different from CPU:
  - if k is a *disk* center then ops_k specifies the number of I/O operations for each visit;
  - if k is a *network* center then ops_k specifies the number of KBytes to be transferred for each visit;

			Routing probabilities $P_{k,h}$															
EQN center	**Number of visits V_k**	**Service rate μ_k (sec^{-1})**	**MSterm**	**MScpu**	**MSdisk**	**WScpu**	**WSdisk**	**PS1cpu**	**PS1disk**	**PS2cpu**	**PS2disk**	**WAN**	**LockWS**	**FreeWS**	**LockPS1**	**FreePS1**	**LockPS2**	**FreePS2**
MSterm	1	1	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0	0	0
MScpu	12	3500	*0.08*	0	*0.75*	0	0	0	0	0	0	*0.17*	0	0	0	0	0	0
MSdisk	9	90	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0	0	0
WScpu	31	16400	0	0	0	0	*0.9*	0	0	0	0	0	0	*0.1*	0	0	0	0
WSdisk	28	420	0	0	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0
PS1cpu	11.5	30000	0	0	0	0	0	0	*0.96*	0	0	0	0	0	0	*0.04*	0	0
PS1disk	11	132	0	0	0	0	0	*1*	0	0	0	0	0	0	0	0	0	0
PS2cpu	11.5	30000	0	0	0	0	0	0	0	0	*0.96*	0	0	0	0	0	0	*0.04*
PS2disk	11	132	0	0	0	0	0	0	0	*1*	0	0	0	0	0	0	0	0
WAN	6	150	0	*0.34*	0	0	0	0	0	0	0	0	*0.5*	0	*0.08*	0	*0.08*	0
LockWS	3	–	0	0	0	*1*	0	0	0	0	0	0	0	0	0	0	0	0
FreeWS	3	–	0	0	0	0	0	0	0	0	0	*1*	0	0	0	0	0	0
LockPS1	0.5	–	0	0	0	0	0	*1*	0	0	0	0	0	0	0	0	0	0
FreePS1	0.5	–	0	0	0	0	0	0	0	0	0	*1*	0	0	0	0	0	0
LockPS2	0.5	–	0	0	0	0	0	0	0	*1*	0	0	0	0	0	0	0	0
FreePS2	0.5	–	0	0	0	0	0	0	0	0	0	*1*	0	0	0	0	0	0

Figure 7. EQN parameters for the *FixedUser* job class

- $capacity_k$ the capacity of center k (in terms of instructions/sec, Kbit/sec or I/O operations/sec, as given by annotation on the DD).

As an example, Table 7 gives the EQN parameters derived for the *FixedUser* job class[4]. In the example case, the number of monitored sites has been limited to only 2. Similar tables are derived by the EQN building procedure for all remaining job classes.

3.3. Method Application

The method application effort can be seen both from the programmer's point of view and from the user's point of view.

From the programmer's point of view, the model building procedure requires a very light programming effort to implement steps 1 through 4, that produce the basic elements of the primordial EQN. A bit larger programming effort is instead required to implement steps 5 and 6 with the `build_EQN` and the `parameterize_EQN` functions.

Regarding step 5, the `build_EQN` function visits in succession all blocks of the EG and for each block introduces a sub-EQN that is then linked to the partial EQN built from the already visited blocks. The function is recursively invoked each time an extended block is visited.

4 It is assumed that the values p_1 through p_n in the Fixeduser EG (see Figure 3) take the values $p_1 = p_2 = 0.5$. It is also assumed that ops_k is equal to 10 KBytes for *network* centers and to 10 I/O operations for *disk* centers.

Regarding step 6, the `parameterize_EQN` function is invoked after the complete EQN is available and only requires using matrix A to derive parameter values according to formulas given in Section 3.2.

From the user's (application designer's) point of view, the EQN model building program can be seen as a tool that can be integrated into a CASE environment and invoked each time decisions are to be taken either to predict the application performance at development time, or to re-design and re-configure the application, or to evaluate its scalability or also to identify critical components. The reader can find in [6] the guidelines to implement the tool according to the XMI standard, to facilitate its integration into a CASE environment.

Conclusions

Performance model building is essential to predict the ability of an application to satisfy given levels of performance or to support the search for viable alternatives. Using automated procedures of model building is particularly relevant in pervasive computing, where the large number of software and hardware components requires introducing models of so large a size that using traditional manual methods of model building would be error prone and time consuming.

This paper has introduced an automated method for building performance models of pervasive computing applications. The proposed method receives as input the UML deployment diagram and the execution graphs of the pervasive application to yield as output the parameterized EQN model, ready to be evaluated by use of any evaluation tool.

By use of the method, the application designer can thus identify and remove bottleneck elements, whether hardware, software or network bottlenecks, and perform sensitivity analysis with respect to the number of fixed or mobile users, the number of deployed sensors and the number of monitored sites.

References

[1] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, E. Cayirci, Wireless sensor networks: a survey, *Computer Networks*, vol. 38, pp. 393–422, 2002.

[2] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-Based Performance Prediction in Software Development: A Survey, *IEEE Transactions on Software Engineering*, vol. 30, n. 5, pp. 295–310, 2004.

[3] V. Cortellessa, A. D'Ambrogio, G. Iazeolla, Automatic Derivation of Software Performance Models from CASE documents, *Performance Evaluation*, 45(2-3):81–106, July 2001.

[4] V. Cortellessa, R. Mirandola, PRIMA-UML: a performance validation incremental methodology on early UML diagrams, *Science of Computer Programming*, vol. 44, pp. 101–129, 2002.

[5] A. D'Ambrogio, G. Iazeolla, Steps towards the Automatic Production of Performance Models of Web Applications, *Computer Networks Journal*, vol. 41, pp. 29–39, January 2003.

[6] A. D'Ambrogio, G. Iazeolla, Metadata-driven Design of Integrated Environments for Software Performance Validation, *Journal of Systems and Software*, Vol 76/2, pp. 127–146, May 2005.

[7] A. D'Ambrogio, SOON: a Tool for Software Performance Validation, *International Journal of Modeling and Simulation*, 2005 (to appear).

[8] A. D'Ambrogio, G. Iazeolla, Design of XMI-based Tools to build EQN Models of Software Systems, *Proceedings of the 23rd IASTED International Conference on Software Engineering (SE2005)*, Innsbruck, Austria, February 2005.

[9] V. De Nitto Personé, G. Iazeolla, The Achilles' Heel of Computer Performance Modeling and The Model Building Shield, in *Computer Performance Modeling: A Perspective*, E. Gelenbe editor, Imperial College Press, 2005 (to appear).

[10] E. Dimitrov, A. Schmietendorf, R. Dumk,UML-based Performance Engineering Possibilities and Techniques, *IEEE Software*, January/February 2002.

[11] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, J. Schiller, Connecting Wireless Sensornets with TCP/IP Networks, *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt, Germany, February 2004.

[12] G. Klyne, F. Reynolds, C. Woodrow, H. Ohto, Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies, available at `http://www.w3.org/TR/CCPP-struct-vocab`.

[13] V. Kumar, S. Parimi, D.P. Agrawal, WAP: Present and Future, *Pervasive computing*, pp. 79–83, January 2003.

[14] Object Management Group, *Unified Modeling Language (UML) Final Adopted Specification*, version 2.0, 2003.

[15] Object Management Group, *UML Profile for Scheduling, Performance and Time*, version 1.0, September 2003.

[16] R. Pooley, P. King, The Unified Modeling Language and Performance Engineering, *Proceedings of IEE Software*, 1999.

[17] C.U. Smith, Performance Engineering of Software Systems, Addison Wesley, 1992.

[18] C.U. Smith, L.G. Williams, *Performance Solutions: a Practical Guide to Creating Responsive*, Scalable Software, Addison Wesley 2002.

[19] M. Weiser, The Computer for the 21st Century, *Scientific American*, vol. 265, 1991, pp. 94–104.