On the Design of a Flexible Software Platform for In-Building OTT Service Provisioning

Tim De Pauw*†, Bruno Volckaert*, Filip De Turck* and Veerle Ongenae†
*Department of Information Technology (INTEC)
Ghent University – IBBT, Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium
†Faculty of Applied Engineering Sciences (INWE)
University College Ghent, Schoonmeersstraat 52, 9000 Ghent, Belgium
Email: tim.depauw@intec.ugent.be

***Abstract*—We propose a software platform which pairs context awareness with over-the-top (OTT) service deployment. By augmenting OTT services with local context information, we allow them to react upon various types of changes in the environment in which they are being deployed. This lets service providers offer more personalized and fine-grained applications, while making use of a third-party infrastructure, via the OTT paradigm.**

Through UML diagrams, we describe the architecture of the proposed service platform. By means of a detailed illustrative scenario, the components involved are further clarified. In addition, in order to prove the feasibility of the architecture, a prototype implementation was developed and deployed on a large wireless sensor network test bed. Using a set of benchmarks, we identified the strengths and weaknesses of both test bed and prototype.

I. Introduction

Recently, the introduction of so-called *over-the-top (OTT)* service providers has given rise to new business models for network-enabled services. OTT providers collaborate with network operators in order to tune the services at hand to the network infrastructure. The most common application of the OTT model is without a doubt found in the digital video domain, where content providers rely on network operators to provide often interactive video content to set-top boxes, television sets and personal computers.

In addition to this classic example of the over-the-top model, the flexibility introduced by its loosely collaborative nature is likely to drive a variety of services backing the future Internet. One factor in such up-and-coming network-based services is *context awareness*, i.e. the ability for an application to intelligently adapt to its changing environment.

In order for a given application to adapt to the environment in which it is operating, context information must be readily accessible. One way of collecting such information is by processing data from hardware *sensors*. In a *wireless sensor network (WSN)*, sensor equipment is generally placed alongside more traditional machines, to collect relevant data.

Once raw data has been collected, knowledge must be extracted from it, so appropriate action can be taken. Oftentimes, this operation is performed via *ontological reasoning*. Ontologies allow for the specification of a semantic model of the problem domain. By consequently applying a reasoner to an ontology, additional information is inferred from the model.

The *WiLab* [1] experimental test bed is a typical WSN. It is comprised of several hundred wireless mesh network nodes, the majority of which are equipped with sensors measuring temperature, humidity, and light intensity. The nodes are embedded systems, modest yet capable of a fair amount of local processing. Thus, the individual nodes' resources can be harvested to drive a distributed context-aware application running on the WSN.

In this paper, we present a software architecture for the operation of context-aware services in WSNs, using the over-the-top paradigm. We also describe a deployment scenario and use it to detail the purpose of the components we introduce. In addition, we list the results of a set of benchmarks, which were used to obtain performance metrics of key components of our platform. These metrics were subsequently used as parameters in resource provisioning algorithms we have developed.

The remainder of this paper is structured as follows. Section II provides an overview of related work. In Section III, we describe the building blocks of our context-aware OTT service architecture, after which Section IV lists our technology choices for its implementation. Section V introduces our evaluation scenario and illustrates how the platform's architecture's components interact. The supporting benchmarks from the used experimental facilities follow in Section VI. We end with a look at opportunities for future research.

II. Related Work

Context awareness and ubiquitous computing are currently important research topics. Ailisto et al. introduce a five-layered model for context-aware applications [2]. Our platform builds upon this model, extending it to support the OTT paradigm.

Ahmad and Begen predict that next-generation networks will be mostly driven by demand for over-the-top video services. An important aspect of these so-called *medianets* will be targeted advertising. As digital media environments become more and more aware of their users and surroundings, content providers will be able to identify micro-scale advertising opportunities. Thus, sophisticated monetization schemes are likely to emerge in the not so distant future. [3]

Ontological reasoning is often used in context-aware applications, as ontologies support specification and processing of context information independent from other application logic.

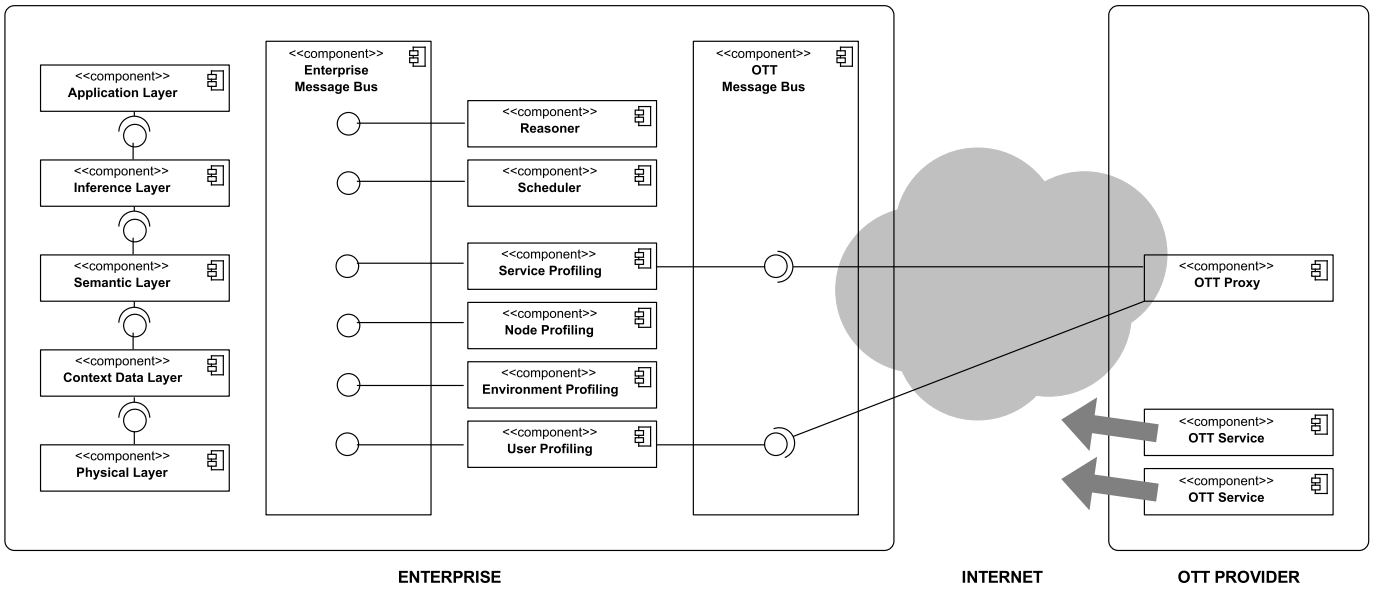

Fig. 1. Component diagram showing the global platform architecture

Originally designed to further the Semantic Web, *OWL* ontologies allow for semantic modeling of entities and relationships. [4] Furthermore, using a reasoner such as Pellet [5], RACER [6], or FaCT++ [7], knowledge can be inferred from ontologies. The *SPARQL* language allows one to query ontology information much like SQL with a relational database.

Pairing ontologies with sensor networks, the *Semantic Sensor Web* enhances the network with spatial, temporal and thematic semantic metadata, resulting in a model unburdened by interoperability issues. [8] This technology is likely to prove useful in a more extensive implementation of our platform.

In light of the resource-intensive nature of ontological reasoning, Verstichel et al. describe a distributed approach. [9] While the concept is certainly compatible with the platform we propose, we chose to focus on key components first and postpone the distribution of reasoning processes.

Component-based distributed applications are an integral part of our context-aware OTT service architecture. While various middlewares are available to support component-based applications, many of these are not suitable for use in a resource-constrained environment such as a wireless sensor network. Arguably the most widely known lightweight Java-based solution is the *OSGi framework* [10]. Various implementations of the standard are available, such as Eclipse Equinox [11] and Apache Felix [12]. Frénot et al. [13] describe an extension to OSGi to support context awareness in distributed environments. A few concepts may be extrapolated to our platform, but we refrain from this so as not to overcomplicate the model.

Our implementation of video playback uses a simple one-way TCP stream. Numerous extensions are imaginable. An advanced application would be the time-shifted multicast solution proposed by Noh et al. [14].

Scheduling tasks in heterogeneous computing environments such as sensor networks is known to be an NP-complete problem. Various researchers have tackled the problem, with mixed results; Dhodhi et al. cite examples of "optimal selection theory-based approaches, graph-based approaches, genetic algorithm-based techniques and other heuristics." [15] Our own bin-packing-based algorithms have been shown to produce acceptable solutions. [16]

III. Software Architecture

Figure 1 shows a high-level architectural view of the platform we propose. It is composed of three main parts.

On the left, the five layers introduced in [2] are displayed. These support the extraction of knowledge from raw data. In Section V, we will illustrate the exact purpose of each layer by means of an example scenario.

A key aspect of the envisaged platform is its *distributed* nature. To allow for maximum flexibility in this area, we introduce a set of loosely coupled *enterprise services*, located in the middle of the figure. These are accessed through two *message buses*. On one hand, internal components use the Enterprise Message Bus to communicate with them; on the other hand, similarly, OTT service providers (on the right) interact with selected services through the OTT Message Bus.

By delegating the location of service components and data to the OTT Message Bus, we can focus on the platform's actual purpose. Similarly, the OTT provider does not need to be concerned with the platform's internals, or even communication with it over the Internet; this task is handled by the OTT Proxy.

Among the enterprise services we propose are four *profiling services*, which maintain knowledge about the network infrastructure, the software components and the users present:

- The Service Profiling service keeps track of OTT services that may be deployed throughout the network. These OTT services are either made available by service providers

or by the network operator. Each OTT service's profile specifies its constraints and requirements in terms of resources. Profiles are dynamic in nature: during operation, the platform monitors the individual services and updates their profiles accordingly.

- Similarly, the Node Profiling service maintains information about the network nodes. A node profile contains information about the node's installed hard- and software, energy conservation schemes, etc. Node profiles, too, may be updated during operation; for instance, memory may be allocated differently, or hardware may fail.
- By means of the Environment Profiling service, the platform keeps track of the physical environment in which the network infrastructure is installed. Environment profiles may for instance list the locations of vending machines, elevators, first aid kits, andsoforth. Evidently, a change in the environment is reflected by a change in its profile.
- Finally, the User Profiling service aggregates information about the persons using the platform and the services installed. While users perform various actions, their profiles are constantly enriched.

Two more enterprise services, the Reasoner and Scheduler support the processes of knowledge inference and task scheduling, respectively.

IV. Implementation Details

In this section, we provide an overview of the tools we envisage for implementation of the proposed software platform. However, while the platform may have been conceived with these tools in mind, its architecture in no way constrains the developer's technology choices.

In terms of hardware, as mentioned, the platform was deployed on our test bed *WiLab* [1]. It consists of 400 *ALIX 3c3* [17] nodes, equipped with a 500 MHz AMD Geode LX800 processor and 256 MB of DDR DRAM. In addition, most of the nodes have a *Moteiv Tmote Sky* [18] sensor board attached, which allows them to measure temperature, humidity, and light intensity. The nodes are equipped with two IEEE 802.11g antennas for wireless communication, while the sensor boards may interact using any IEEE 802.15.4 protocol.

Software-wise, we used the following technologies:

- WiLab nodes have *Voyage Linux* [19] installed, an embedded derivative of the Debian GNU/Linux distribution.
- The *OSGi framework* [10] offers a lightweight solution for component-based Java development. Our implementation of choice is *Eclipse Equinox* [11].
- To manage ontologies in Java, we used the *Jena Ontology API* [20]. One of the advantages of this common library is its straightforward integration with the *Pellet* reasoner [5], which we also deployed.
- To transcode video material, we relied on *FFmpeg* [21].
- Video files were streamed using the headless version of the popular *VLC media player* [22].
- Scheduling the deployment of OTT services is done using algorithms based on the bin packing metaphor; these were previously published in [16].

V. Illustrative Scenario

Having introduced the prime components of our platform, let us now describe a typical scenario in which it is applied.

Say an OTT provider wants to run a video advertisement for a refreshing new soft drink. To this end, a wireless sensor network not too dissimilar from our test bed is installed, and video screens are installed near vending machines. The OTT provider pushes his video advertisement service to our platform. Based on context information provided by the platform, the OTT service detects advertising opportunities; for soft drink ads, a rising temperature may be an excellent incentive. Once such an opportunity manifests itself, the platform provisions resources for the deployment of the advertisement. For instance, steps may be taken to play the ad on the video screens, or even the mobile devices of the users present, accompanied by a personalized coupon code.

Now, we will analyze this real-life scenario and pinpoint the components involved. By means of sequence diagrams, we will clarify the interaction between the various parties.

A. OTT Service Registration

First of all, the OTT service provider registers a new OTT service with the platform, e.g. the soft drink advertisement service. The corresponding sequence diagram is shown in Figure 2. Via the OTT Message Bus, information about the newly registered OTT service is made available to the platform, and then propagates to the classes involved:

- A call to Service Registration ensures that the platform becomes aware of the existence of the OTT service.
- If the service provider did not supply a profile with the OTT service, the platform performs an initial analysis of the executable code supplied.
- The service profile is subsequently registered with the Service Profiling service.
- Finally, the OTT service's executable code is placed in the Service Repository, rendering it available for deployment on the network.

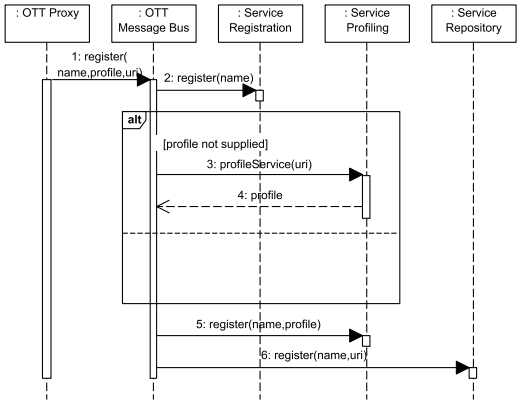

Fig. 2. Sequence diagram detailing OTT service registration

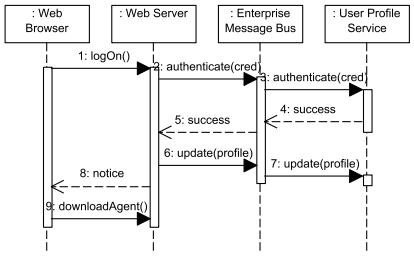

Fig. 3. Sequence diagram detailing session initiation

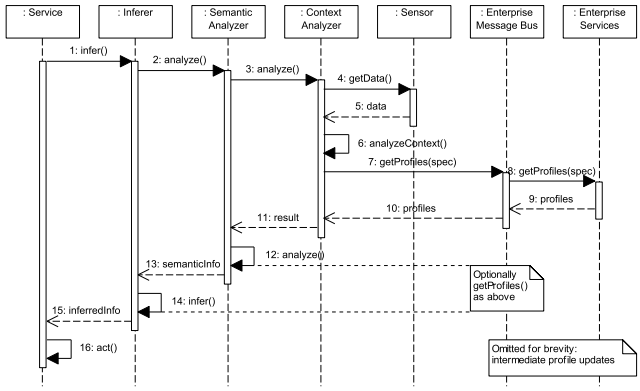

Fig. 4. Sequence diagram detailing *pull-based* OTT service invocation

B. OTT Service Activation

After the registration of an OTT service, the service provider may activate or deactivate it at will. This subscenario is a simple application of the OTT Message Bus. The OTT Proxy contacts the bus, which forwards the request to the Service Profiling service. Thus, as already outlined in Section III, the distributed nature of the platform barely imposes additional complexity regarding OTT service management.

C. Session Initiation

Having registered and profiled the OTT service, the platform is now ready to receive users. As mentioned, we want to be able to interact with them: not only do we want to detect their presence in a certain area and whatnot, but ideally, we would also like permission to push useful information and advertisements to their mobile device. Therefore, we wish to be able to install a software *agent* on it. Figure 3 shows this.

It is assumed that the user has received account information prior to session initiation, or, alternatively, that anonymous logins are allowed. Upon access to the platform, the User Profile Service is contacted to store new information about the user and his device. Consequently, a Web page is served, containing a link to the device-specific installer for our software agent. Upon the installation of this agent, the device becomes capable of sharing context information with the platform, as well as receiving rich media from it.

D. OTT Service Invocation

Ultimately, of course, our goal is to invoke the video advertisement OTT service. This final step in our scenario is therefore probably the most interesting one. It embodies the transformation of sensor data to context information, as well as the inference of knowledge and the subsequent response.

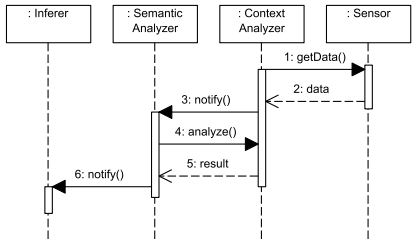

Fig. 5. Sequence diagram detailing *push-based* OTT service invocation

In Figure 4, we illustrate the case where the OTT service, located in the Application Layer, takes the initiative to *pull* updates from the underlying layers. Thus, from left to right, the first five lifelines in the diagram correspond to the five layers in Figure 1, whereas the rightmost lifeline, Enterprise Services, is a placeholder for the platform's profiling services.

Using our example scenario, let us walk through the architecture's five major layers, so we can obtain a clear insight in how the raw sensor data is transformed.

1) Responding to the getData() message, the Physical Layer embodied by a Sensor provides the raw measurement `301` to the Context Data Layer. The fact that this number actually represents a temperature expressed in Kelvin is unknown at this point. Another example may be a list of mobile devices that are currently in the sensor's vicinity.
2) The Context Data Layer is embodied by the Context Analyzer, which builds a local representation of context. In our example, this boils down to obtaining a more accurate room temperature by computing an average of several sensor readings. Which sensor readings to include is determined by consulting an environment profile, meaning that the Context Analyzer must communicate with the enterprise services. Having obtained the necessary profile, it uses the additional measurements `298` and `310` to compute an average temperature of 30 degrees Celsius. In the case of positioning, user profiles may be obtained to produce a list of users in the room.
3) The Semantic Analyzer, part of the Semantic Layer, takes the resulting value and analyzes it further. In this case, 30 degrees is taken to be "quite high," prompting an opportunity to play the OTT provider's ad. When dealing with the users present, their profiles may be analyzed to see if they are part of the target demographic for the ad.
4) Having been notified of the advertisement opportunity, the Service in the Application Layer acts upon it. For instance, it may contact the Scheduler to provision resources for streaming video to the appropriate devices.

Sometimes, it may be more efficient for intermediate layers to produce *events*, which are then *pushed* to the layer above. This scenario is exemplified by Figure 5, where the Context Analyzer eventually triggers an update to the topmost layer.

TABLE I
UOB SPARQL QUERY EXECUTION TIMES

Query	Time
1	6,536 ± 152 ms
2	12,209 ± 137 ms
3	4,726 ± 120 ms
4	1,552 ± 55 ms
5	5,209 ± 120 ms
6	1,963 ± 60 ms
7	8,657 ± 142 ms
8	–
9	2,405 ± 61 ms
10	1,711 ± 45 ms
11	5,808 ± 108 ms
12	993 ± 24 ms
13	–

(a) OWL Lite

Query	Time
1	3,906 ± 42 ms
2	10,770 ± 150 ms
3	12,176 ± 132 ms
4	7,095 ± 94 ms
5	14,433 ± 153 ms
6	13,171 ± 142 ms
7	10,101 ± 122 ms
8	–
9	9,371 ± 120 ms
10	9,417 ± 107 ms
11	7,551 ± 90 ms
12	19,180 ± 154 ms
13–15	–

(b) OWL DL

VI. TEST BED BENCHMARKS

To accurately provision resources on our test bed, the platform must be aware of various characteristics of the hard- and software involved. In this section, we describe the benchmarks we used to compose service and node profiles.

A. Reasoning Throughput

While conceptually, ontological reasoning is well suited to the context awareness problem domain, unfortunately, working with ontologies quickly becomes a resource-intensive operation. Consequently, to quantify the capabilities of our test bed regarding ontologies, we used the well-known *University Ontology Benchmark* [23].

It quickly became apparent that the larger variants of the benchmark, containing information about 5 and 10 fictitious universities, respectively, are too complex for analysis on a single node. Both with an in-memory model and a TDB-based one, we were unable to load the full ontology. This is not surprising, as far more powerful machines have been known to struggle with the benchmark as well. [24]

The single-university version of the benchmark, however, did provide us with some insight. For the OWL Lite flavor, on average, it took Jena and Pellet 131 seconds to load the full ontology from flash storage and infer additional facts; in the OWL DL case, this process required 137 seconds. Subsequently applying the supplied set of SPARQL queries to the model yielded mixed results; the means and variances of 20 runs are displayed in Table I. As can be observed, certain queries did not complete properly. In addition, query 11 produced a result set different from the reference solution in both cases; this behavior, too, has been documented in [24].

In the case of OWL Lite, memory usage peaked at 96 MB, whereas the OWL DL variant required 103 MB of RAM. Both these values solely apply to the ontological reasoning thread, not the Java Virtual Machine itself. As our nodes only have a total of 256 MB of RAM, it is safe to assume that they are unable to perform reasoning tasks concurrently. On the other hand, we can also conclude that the nodes are capable of a fair amount of ontology processing within acceptable time.

B. Video Transcoding Performance

We envisage two scenarios for adapting a video clip to various device profiles. In the first scenario, a video clip is transcoded using a set of profiles for common devices, yielding a collection of video files. When a video is to be played back, the most appropriate file is selected and streamed to the devices. The second scenario transcodes any given video clip or stream in real time and immediately streams it to one or more devices. It goes without saying that this is a time-critical operation, but it does of course offer more flexibility.

To assess our test bed's capabilities in terms of video transcoding, we used FFmpeg to transcode three video files of varying qualities, in an attempt to represent popular choices. The characteristics of these files are given in Table II. The files were transcoded to two formats, each time at minimal, half and maximal quality. The formats we chose were *Dirac* and *Ogg Theora*. We had hoped to include H.264 as well, but were unable to get the required software functioning; the Dirac codec is however assumed to provide similar performance. The results of the benchmark are summarized in Table III, in which ratios were calculated by dividing the time required for transcoding by the video's duration.

With none of our benchmarks approaching real-time transcoding (i.e., a ratio of 1 or less), even without any transformations such as scaling, we must conclude that our hardware does not meet the requirements for combined transcoding and streaming. Streaming itself, however, does not pose a problem; we verified this by streaming the same files using VLC media player, without any transcoding applied.

The transcoding operation must consequently be carried out in advance. Either the OTT provider must therefore supply video files adhering to the platform's requirements, or a transcoding service must be added to the platform.

C. Scheduling Performance

The measurements above formed the basis of a test case for the scheduling heuristics we published in [16]. We used time slots representing 10 seconds of wall clock time. Starting from 200 WiLab nodes divided into 6 zones, and assuming that each

TABLE II
VIDEO FILE CHARACTERISTICS

File	Video Codec	Video Bitrate	Video Resolution	Audio Codec	Audio Bitrate
1	MPEG-1	1,098 kbps	352×240	MP2	112 kbps
2	H.263	2,228 kbps	640×352	MP3	160 kbps
3	H.263	5,849 kbps	1,280×720	PCM	1,536 kbps

TABLE III
VIDEO TRANSCODING RATIOS BY FORMAT AND QUALITY

File	Format	Minimum	Half	Maximum
1	Dirac	22.292	27.197	29.177
	Ogg	2.947	7.033	4.641
2	Dirac	48.634	114.032	60.878
	Ogg	6.148	7.609	8.211
3	Dirac	38.981	38.824	105.500
	Ogg	27.115	38.572	34.557

TABLE IV
SCHEDULED TASK TYPES

Type	Amount	Memory Usage	Duration
Reasoning	50	150 MB	13–15 slots
Streaming	50	150 MB	3–6 slots
Miscellaneous	50	50–100 MB	1–5 slots

TABLE V
RESULTING SCHEDULE METRICS

Heuristic	Execution Time	Loss	Nodes Used
First Fit	31.53 ± 11.77 slots	6.53%	99.33 ± 14.04
Next Fit	31.60 ± 11.74 slots	6.74%	102.68 ± 10.93
Best Fit	31.40 ± 13.25 slots	6.15%	99.62 ± 9.61
Fair Fit	32.29 ± 14.05 slots	8.73%	121.16 ± 14.14

node already had a number of tasks assigned, we scheduled additional tasks on the network, with a minimization of the combined execution time as the objective. In what follows, all random values have a discrete uniform distribution.

The tasks which were previously assigned did not overlap and were between 5 and 20 slots apart; their duration varied between 1 and 10 slots and they used 50 to 150 MB of RAM, 155 MB being the total amount available. We then proceeded to schedule 3 types of tasks, as summarized in Table IV. Each task was randomly tied to one of the 6 zones. "Miscellaneous" tasks, which represent any sort of task carried out by an OTT service, were made dependent on a randomly chosen task with a lower number; this enforces an order relation.

We ran these simulations 100 times on our test bed WiLab, using Java 6. The resulting combined execution times—i.e., for all the tasks—are presented in Table V. The average loss compared to the optimal execution time is also displayed; optimal schedules were obtained by means of integer linear programming [25], using far more powerful hardware.

Like in [16], the Best Fit heuristic lives up to its name and slightly outperforms its peers. For our type of problem, Fair Fit appears to be the worst choice, both in terms of combined execution times and node occupancy; we do however note that its time loss of 8.73% is still a fairly competitive result.

On average, the four solvers executed in a mere 790 ms and required just 24.58 MB of RAM. These modest requirements, combined with the acceptable results which the heuristics produce, render them highly suitable for real-time use on our resource-constrained test bed.

VII. CONCLUSIONS AND FUTURE WORK

We described in detail an architecture for enabling context awareness in OTT applications. Using an illustrative scenario, the components involved were subsequently discussed. Prototype deployment on our test bed allowed us to carry out various benchmarks, which evaluated the feasibility of the platform. Thus, we have shown how our proposed architecture can support context-aware OTT services, as well as proven that our test bed may be used as a deployment environment for this architecture. At the same time, we have identified some limitations of the test bed. Based on our findings, we will now focus on the implementation of more extensive scenarios.

ACKNOWLEDGMENT

Tim De Pauw would like to thank the University College Ghent Research Fund for financial support through his Ph.D. grant. Part of this work has been funded by the IWT SBO SymbioNets project.

REFERENCES

[1] L. Tytgat, B. Jooris, P. De Mil, B. Latré, I. Moerman, and P. Demeester, "Demo abstract: WiLab, a real-life wireless sensor testbed with environment emulation," in *European conference on Wireless Sensor Networks (EWSN)*, Cork, Ireland, Feb. 2009.

[2] H. Ailisto, P. Alahuhta, V. Haataja, V. Kyllönen, and M. Lindholm, "Structuring context aware applications: Five-layer model and example case," in *Workshop on Concepts and Models for Ubiquitous Computing*, 2002.

[3] K. Ahmad and A. Begen, "IPTV and video networks in the 2015 timeframe: The evolution to medianets," *IEEE Communications Magazine*, vol. 47, no. 12, pp. 68–74, 2009.

[4] D. L. McGuinness and F. van Harmelen. (2004, Feb.) OWL Web Ontology Language Overview. [Online]. Available: http://www.w3.org/TR/owl-features/

[5] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical OWL-DL reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, Jun. 2007.

[6] V. Haarslev and R. Müller, "RACER system description," *Automated Reasoning*, pp. 701–705, 2001.

[7] D. Tsarkov and I. Horrocks, "FaCT++ description logic reasoner: System description," *Automated Reasoning*, pp. 292–297, 2006.

[8] A. Sheth, C. Henson, and S. Sahoo, "Semantic Sensor Web," *IEEE Internet Computing*, pp. 78–83, 2008.

[9] S. Verstichel, F. Ongenae, B. Volckaert, F. De Turck, B. Dhoedt, T. Dhaene, and P. Demeester, "An autonomous service platform to support distributed ontology-based context-aware agents," *Expert Systems: The Journal of Knowledge Engineering on Engineering Semantic Agent Systems*, Accepted for publication.

[10] OSGi Alliance. (2003) OSGi Service Platform, release 4. [Online]. Available: http://www.osgi.org/

[11] Eclipse. Equinox. [Online]. Available: http://eclipse.org/equinox/

[12] Apache Foundation. Felix. [Online]. Available: http://felix.apache.org/

[13] S. Frénot, N. Ibrahim, F. Le Mouël, B. Hamida, J. Ponge, M. Chantrel, and D. Beras, "ROCS: a remotely provisioned OSGi framework for ambient systems," in *2010 IEEE Network Operations and Management Symposium (NOMS)*, 2010, pp. 503–510.

[14] J. Noh, A. Mavlankar, P. Baccichet, and B. Girod, "Time-shifted streaming in a peer-to-peer video multicast system," in *2009 IEEE Global Telecommunications Conference (Globecom)*, 2010, pp. 1–6.

[15] M. Dhodhi, I. Ahmad, A. Yatama, and I. Ahmad, "An integrated technique for task matching and scheduling onto distributed heterogeneous computing systems," *Journal of Parallel and Distributed Computing*, vol. 62, no. 9, pp. 1338–1361, Sep. 2002.

[16] T. De Pauw, S. Verstichel, B. Volckaert, F. De Turck, and V. Ongenae, "Resource-aware scheduling of distributed ontological reasoning tasks in wireless sensor networks," in *2010 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, Newport Beach, USA, 2010.

[17] PC Engines GmbH. ALIX system boards. [Online]. Available: http://pcengines.ch/alix.htm

[18] Sentilla Corp. Moteiv Hardware Product Transition Notice. [Online]. Available: http://www.sentilla.com/moteiv-transition.html

[19] Voyage. Voyage Linux. [Online]. Available: http://linux.voyage.hk/

[20] I. Dickinson. (2009, Feb.) The Jena Ontology API. [Online]. Available: http://jena.sourceforge.net/ontology/

[21] FFmpeg. [Online]. Available: http://ffmpeg.org/

[22] VideoLAN. VLC. [Online]. Available: http://www.videolan.org/

[23] L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu, "Towards a complete OWL ontology benchmark," vol. 4011, pp. 125–139, 2006.

[24] M. Luther, T. Liebig, S. Böhm, and O. Noppens, "Who the Heck is the Father of Bob?" *The Semantic Web: Research and Applications*, pp. 66–80, 2009.

[25] R. J. Vanderbei, *Linear Programming: Foundations and Extensions*, 3rd ed. Springer, Nov. 2008.