On the efficiency of semantic web based context computing infrastructures

Alberto Garcia-Sola
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia
Murcia, Spain.
agarciasola@um.es

Teresa Garcia-Valverde
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia
Murcia, Spain.
mtgarcia@um.es

Francisco Lopez-Marmol
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia
Murcia, Spain.
pacolopez@um.es

Juan A. Botia
Departamento de Ingeniería de la Información y las Comunicaciones
University of Murcia
Murcia, Spain.
juanbot@um.es

Abstract − *We present in this paper the results of testing the OCP2 middleware. OCP2 is an efficient middleware for context information management with reasoning capabilities based on semantic web, which offers different ways to express real world behavior (such as rules or patterns), modeling them on top of the ontology. More specifically, we focus on its performance and scalability. Performance has always been one of the biggest drawbacks of the semantic web, and we have work hard to change this. The results are outstanding marks using hundreds of simultaneous users.*

Keywords: OCP, semantic web, OCP2, ontologies, context, efficiency, rules, patterns

1 Introduction

Traditionally, Semantic Web has been mainly used in research areas. Applications which require heavy-usage and immediate user response usually prefer different alternatives, such as traditional databases, basically due to the inefficiency of the former tools to deal with Semantic Web. Some minor projects have used ontologies for commercial applications, but usually, they consisted on small projects. Efficiency on Semantic Web is a traditional known issue that should be solved in order to become a spread technology. Context-Aware applications are usually based on ontologies and semantic web, but to be fully usable in any scenario they need stability and efficiency.

Our proposal is this middleware tuned up to achieve high performance. In this second version of OCP (which stands for Open Context Platform), a lot of improvements have been done, not only have we focus on efficiency, but also in reliability and functionality. The first OCP version [10] was a very good proof of concept, and has been used in diverse real projects, however, it lacked of the mentioned efficiency. OCP2, on the contrary, is highly efficient, being able to process thousands of events reasoning with them in a fraction of second. It is also easier to use than OCP first version and have some new characteristics such as patterns or rules which make it a very useful tool to deal with Context Aware scenarios.

In section 2 we discuss about other works related with context reasoning. In the following section (3) we describe the OCP Middleware. Section 4 describes the OCP context reasoning module is described. In Section 5 we can see the OCP2 performance through some experiments to measure its capabilities to end up with conclusions and future work (sections 6 and 7).

2 Related Work

OCP is related with many architectures in different ways. The oldest architecture related with OCP could be the blackboard model, proposed by B. Hayes-Roth [7]. OCP has this three elements of the blackboard model (Knowledge Sources, Control Shell and Blackboard) and are organized in a similar way. However, OCP is more focused on a middleware concept, with an indirect and anonymous communication [5]. A similar processing model with no AI charge is that of the data centered publish/subscribe model (DCPS), where

there is a central data repository and a set of producers and consumers that produce and consume data, respectively. There is a *data model* that defines how data is represented. A try to standardize this model is [11]. Active Databases [4] with languages such as Snoop and Ode can specify events, as OCP does in a different way with patterns. They are traditional databases provided with ECA pattern management. Closer to the OCP pattern recognition are the CEP (*Complex Event Processing*) systems. OCP can be seen as a semantic CEP. This has been studied since 2003 [12].

First OCP version dates back to 2006 [10]. Since then to the current OCP version (OCP2) system has evolved and improved in many aspects (e.g. ease of use, distributed, history, reasoning...). However, context-aware systems are an increasing trend in the last few years. There are several platforms and middlewares for context management with diverse approaches. In [14, 8] we can find a detailed survey of some of them, including different approaches (e.g. object-oriented models, ontology based models, key-valued models...), to conclude that ontologies are the more promising alternative. [8] offers a different view centered in journals from 2000 to 2007. Numerous context-aware articles have grown considerably since 2000, up to 7 times more in 2007.

Regarding the use of reasoning in these systems, there is also a wide range of works. In [2] we can find a survey discussing many of them. A variety of context reasoning schemes have been proposed to deal with context reasoning, including Bayesian networks [9], Dempster-Shafer Theory [16], logic-based [1] and ontology-based context reasoning schemes among others, but only ontology-based reasoning schemes incorporate semantics into context representation and reasoning. Evolving from semantic networks, ontology provides a structural model that enables applications to handle complex and disparate information.

3 The OCP Middleware

OCP is a middleware for managing context information with the following characteristics.

Event oriented. In OCP, there are two roles, the producers and the consumers of context. Context producers can send information at any time, so that context consumers interested in this information will immediately receive it. Consumers can set their own interests to receive context updates of some concrete entities that interest them, or even all instances of an entity type.

Distributed. OCP offers a client/server architecture. The client only needs a few lines of code to communicate with the server and handle all context. OCP server is responsible for accomplishing reasoning and distribution work for the clients, allowing interaction with lightweight devices (i.e. sensors and actuators).

Ontology oriented. Context information is organized and stored using OWL ontologies. Ontologies are a powerful tool for knowledge representation, and there are several authors who describe them as the most promising tool to represent context information [14]. Two ontologies are used in OCP, context ontology and domain ontology. Domain ontology includes every element (either structural or instance) of the domain that we are working in. All that is susceptible to be handled in the system (i.e. in terms of reasoning, history, etc.) must be represented in this ontology. The context ontology gives us some useful information for OCP to manage context, including an identifier, sequence number and timestamp. The only restriction of the domain ontology is that All entities must inherit from this OCP Context ontology (*ContextEntity*). This allows us to treat them uniformly and that OCP automatically save some useful information (e.g. timestamp). Using an ontology as substrate allows us to abstract our customers much more than a simple attribute-value system as Context Toolkit [13], specifying only the specific information they need, despite there will be more information underneath than just this concrete relation. For instance, consumers may be interested in changes of *Person*, and other in changes of *Patient*, being *Patient* subclass of *Person*. However, if we introduce information about *Patient*, both will be notified. Furthermore, ontologies allow the user to reuse well-known ontologies rather than start from scratch, and adapt to their needs. OCP is extensible with already defined ontologies, e.g. SOUPA, CONON, ULCO, etc.

Extensible. Derived from the previous one. By using a domain ontology defined by the user, the system applies to any domain. The fact of using ontologies allows us, in turn, knowledge reuse using well-defined ontologies of different domains; we can compose large-scale context ontologies without starting from scratch. Instead of offering a larger ontology with certain preset elements as is done in [15], we prefer to let the user define it, and, if desired, reuse other ontologies. The user must only import the aforementioned *ContextEntity*.

Context history. The context history is a powerful and flexible OCP tool that stores a historical of every context information (or what you specify) in the system. This allows us to perform temporal reasoning and learning through adaptation as well as perform data mining using historical data through the API that offers OCP. Depending on the system, it is possible we do not want to store all information. For this, OCP offers different storage modes:

- Store nothing.
- Store everything.
- Store only new events. Thus, OCP would only store events involving a change of context. If we have a sensor that every certain time tells us the temperature, you may be interested in only keeping

temperature changes, and ignore the values which represent no change.

- Store intervals. Unlike the former, we store intervals, so we know that for some time the same value has been sent and the start and end during which this value was sent.

Context reasoning. OCP includes two ways of reasoning, endowing the system with the capabilities of inferring new information. This way OCP clients stick to send and receive information in a way that they understand, letting OCP the process to give meaning to that information. OCP uses both rules and pattern recognition. This reasoning is fully depicted in section 4.

Thanks to OCP we can, with a few lines of code have full control over the system context, and even infer not explicit new information.

4 Context Reasoning within OCP

OCP is a middleware that allows us to control the context in a simple and distributed way between producers and consumers. The use of semantic context information based on a predefined domain ontology gives us great expressiveness and versatility. In general, the producers' logic is limited to generate data, and send it to the middleware regardless of the ontology, being the middleware responsible for providing it a semantic meaning within the domain. However, it is possible that this data in that format is not relevant to consumers, needing reasoning on it to make explicit certain implicit information. OCP performs this reasoning, thus freeing both consumers and producers of the need to include more complex logic and gather information irrelevant to them, focusing on things important to them. Thus, domain logic (i.e. any relevant information for the applications about the entities in the environment and their relations) is centralized in a single point (i.e. OCP). Take, for example a temperature sensor. The sensor just generates temperature data. Depending on the domain, a temperature of 36 C may mean different things. If we are measuring a patient temperature, it may mean that the temperature is correct, if we are measuring a room temperature it may be too high and we should turn the air conditioner on. If it is used for fire detection, it will mean something different. However, the sensor is unaware of this. It simply has to send the temperature data. OCP is the one who gives meaning to this information using reasoning, implicit reasoning (axiomatic) and explicit (rules and patterns).

In Semantic Web based representations, part of this new information generated by reasoning can always be obtained by the so-called "axiomatic reasoning", using certain implicit and universal rules for all ontologies. For instance, if we define the relation *equalsTo* as symmetric, when we add the relation *class1 equalsTo class2*, we implicitly know that *class2 equalsTo class1*. However, there is some domain information that must be described in the form of rules or patterns. These rules are needed to provide this new information, since they are domain dependent, and, possibly, dynamic (may change over time). OCP supports two types of reasoning: to detect and correct inconsistent context information and to derive higher level context information. The latter type of reasoning is based on properties like symmetry (as aforementioned) and transitivity and user-defined rules or patterns. We will focus on the latter type of reasoning, intended to derive new information.

Rules are a flexible way to express specific domain information. However, it is not always the best and easiest way to express this information. Patterns offer us a way to express this information on the form of some events that must match in the system in order to infer this new information. It is easier to express temporal clauses as well as causality between entities.

For this reasoning we use ORE-API [1]. ORE-API is a set of tools developed in our lab for reasoning using Semantic Web used by ORE (Ontology Rule Editor), graphical tool used for easy ontological rule sets creation, debugging and validation. It is a powerful tool that allows with few lines of code and in a simple way read, modify and create new rules as well as reason information using different methods. We chose to use an API instead of reasoning tools such as Pellet directly. This approach has advantages and disadvantages. On one hand, ORE-API offers various types of reasoning (combinations of different reasoners), possibility to modify the rules easily, great versatility, updates and has been tested in different environments. Furthermore, using ORE-GUI we can create/view/modify graphically the different rules used in OCP. ORE-API also offers a powerful and easy programming interface to read, create and modify rules in different formats and export capabilities. All reasoning logic is encapsulated, so that any change is made simple. However, the election of ORE-API also has some drawbacks, since it creates an extra layer of abstraction with a possible performance penalty.

When OCP does not use the reasoner, the operation flow when new context is notified by a producer is the following: (1) Receive and manipulate the message. (2) Insert/modify ontology information. (3) Update the history. (4) Notify concerned consumer. Reasoning entail the possible generation of new knowledge, to be updated itself into the historic and possibly notified to different consumers. Since new information is needed to be in the ontology to reason with it, the only temporal constraint is the reasoner to be called after the insertion/modification of information in the ontology. In our case, we have decided to make this reasoning after the first notification to consumers for several reasons.

[1] http://ore.sourceforge.net/

On the one hand, we avoid unnecessary waiting for the consumers to be notified, probably not interested in the new created context. On the other hand, this way there is some sequence (also in history) about events, giving priority to information received prior to that generated, avoiding cases of consumers who interpret the generated information before producing potential inconsistencies. Note that the new inferred information should be introduced in OCP so that it can notify potential consumers and generate the historical of this information, which will be treated as information generated by a producer for all purposes.

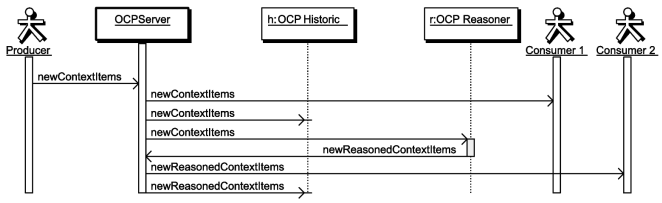


Figure 1: New context arrived to OCP. Sequence diagram.

In figure 1 we can see a sequence diagram of how OCP handles a new context event. The first thing done by OCP when receiving new context information (*newContextItems*) is notifying the subscribed consumers, as discussed previously. Then, reasoning is performed on the new information received, and if any new knowledge is inferred (*newReasonedContextItems* in the figure), it is sent to consumers (if any subscribed to that new information).

Reasoning in OCP is compound of two parts, the axiomatic reasoning and the rule-based reasoning. From the range of options offered by ORE-API, we have used Jena (Rule-based Reasoning Engine) + Pellet (as Ontology Reasoning Engine) reasoning by default. On the one hand, there is the Ontology Reasoning, carried out by Pellet. Pellet is an open-source Java OWL DL reasoner based on the tableaux algorithm. Other approaches such as Bossam (translate OWL-DL into rules and give the rules to a forward chaining engine) do not fully cover OWL-DL (cannot cover the full expressivity of OWL-DL due to many incompatibility between Description Logic and Horn Rule formalisms). Moreover, Pellet integration with Jena (ontology management is done in OCP using Jena) and ORE-API is straightforward. Knowledge models based on DL ontologies are usually divided into TBox (terminological) and ABox (assertional) components [6]. The TBox contains the vocabulary and schema that define domain concepts, their properties, and the relationships (called roles in DL) among them. Therefore, once OCP has started, TBox never change, since the schema remains unchanged, and reasoning over TBox must only be done once. However, ABox, which requires a previous TBox reasoning does change through time, hence, reasoning to ABox must be done every time a new item arrives the system.

On the other hand, there is the Rule-based Reasoning, carried out by Jena. The selected rule reasoners should provide the knowledge inferred by the rules isolated from the rest of the knowledge inferred by other reasoning processes. This feature is required by OCP to be aware of the new information and let it process properly, as well as for debugging tasks. The Jena rule reasoner makes this distinction and split the inferred knowledge from the base knowledge whereas the Pellet rule reasoner does not. Jena Rule-based Reasoning has diverse built-in engines, which support forward chaining, tabled backward chaining and hybrid execution strategies [3]. Rules have been introduced in the ontology using the Semantic Web Rule Language (SWRL) [2] based on a combination of the OWL-DL and OWL-Lite sublanguages of the OWL with the Unary/Binary Datalog RuleML sublanguages of the Rule Markup Language. These rules has a semantic equivalence to well researched description logics. SWRL allows us a great power and expressiveness. Its ability to write our own built-ins allows us to model new functionality to interact with the history and perform temporal reasoning in a more natural way.

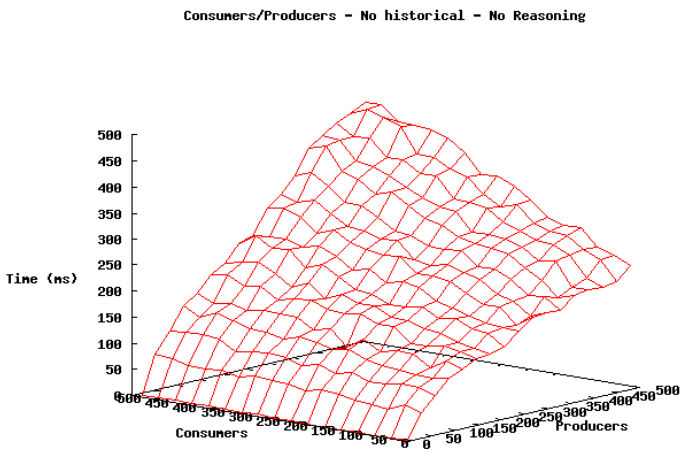
5 Efficiency on OCP

In this section OCP2 is evaluated in order to test its performance in different scenarios. We are looking for the influence of number of producers, consumers and different configurations of ontologies with its complexity. We have used ontologies expressed in OWL and the latest OCP2 version running into a 64 bits Java Virtual Machine.

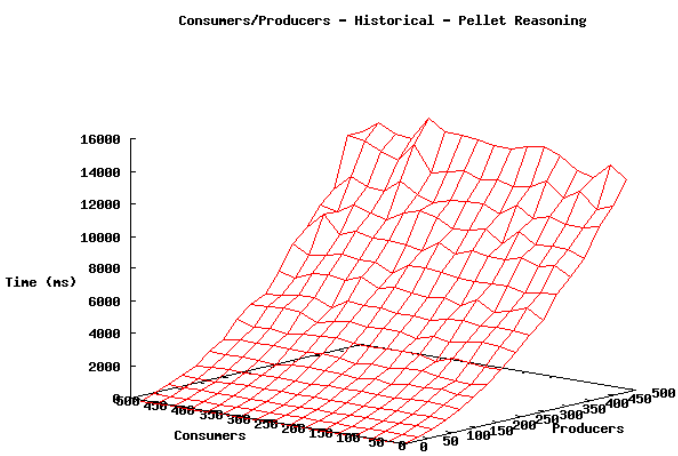
Experiments consist on measuring the time spent by OCP and the clients depending on the configuration. All the producers synchronize and send an item at the very same moment, to bring OCP to a simulation of a very extreme situation. For every configuration, producers can do a number of operations to get an average. Furthermore, that operations can also influence in the results, since the response of the server after being used may be different from that of just starting, since it may contain information which has to be processed and can be related with the results. Therefore, after every experiment, the server is restarted and the knowledge database reset. For every experiment done, we have measured these times:

First producer - last consumer. This is the time spent since the first producer sends the first item of information until the last consumer receives the last item. If there are 3 consumers subscribed to the information produced by the producers, and 5 producers, every consumer receives 5 items of information per operation, and every producer produces 1 item per opera-

[2] http://www.w3.org/Submission/SWRL/



Test 1.1.- No historical - No reasoning



Test 1.5.- Historical - Pellet reasoning

Figure 2: Test 1 - Consumers/Producers experiments

tion. There this time measures from the first item sent until the 15 items are received by all the consumers, including network time and OCP Server internal processing time.

Total OCP processing. Time spent internally in OCP, since it receives an item until it sends it through the network to the listeners (if the case).

Internal OCP processing. Time spent internally in OCP, excluding reasoning time. When reasoning is deactivated, is equal to the Total OCP processing.

Network time. Time spent since the message it sent in OCP until it is received by the clients.

Reasoning time. Time spent by the reasoner (if active) trying to infer new information.

All tests have been done in the same machine, using it for OCP Clients (i.e. producers and consumers) and OCP Server. Therefore, memory usage and CPU usage has been higher than it will be in scenarios where OCP server will be a host apart from the clients. We have used a Thread for every client. Experiments have been limited in size due to this, since thousands of Threads require a high amount of RAM. However, we have used this decision since it is much easier to synchronize Threads to make them start at the very same moment and measure the times accurately.

In order to properly evaluate OCP2 we have done an initial evaluation of what could influence OCP. We found these dimensions:

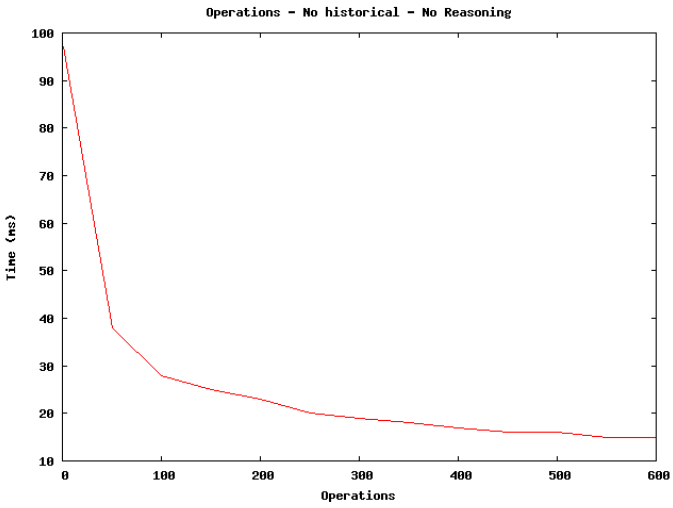
- Number of individuals within the ontology.
- Ontology schema size (number of entities).
- Present rules/patterns in the ontology.
- Fired rules/patterns.
- Number of producers.
- Number of consumers.
- Number of operations per producer.
- Historical on / Historical off.
- Chosen reasoner / Reasoner off.

Due to the number of different dimensions is was unfeasible to vary all of them, so we have limited the variation of them. Tests are done keeping some of them constant and varying the other.

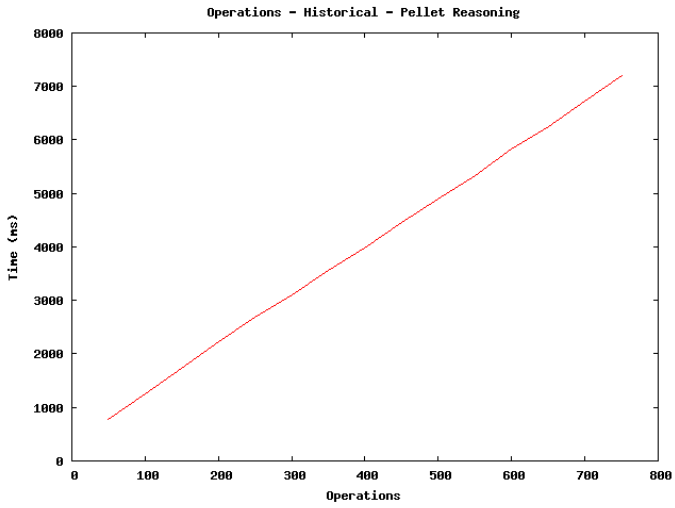
We created some initial experiments with some fixed values for every dimension depicted above and changing the values for a concrete dimension, to see how did it behave. Most of the information was not far from our estimations. This information has been useful to define the following experiments, which are the most representative.

Test 1 (see Figure 2). Number of Consumers/Producers. This experiment involves different smaller experiments. We have set up different reasoning configurations, using or not the historical. The ontology as well as the individuals within it have been kept simple. The ontology contained two rules, one of them which was fired, and the other present but never fired. Every single *producer* generates 5 items (operations). The number of both consumers and producers goes from 1 up to 500. Besides, we have launched six different configurations of this experiment:

- Test 1.1. No reasoner. No rules. Historical off.
- Test 1.2. Pellet reasoner. Rules activated. Historical off.
- Test 1.3. Jena-full reasoner. Rules activated. Historical off.
- Test 1.4. No reasoner. No rules. Historical on.



Test 2.1.- No historical - No reasoning



Test 2.5.- Historical - Pellet reasoning

Figure 3: Test 2 - Operations experiments

- Test 1.5. Pellet reasoner. Rules activated. Historical on.
- Test 1.6. Jena-full reasoner. Rules activated. Historical on.

Since the number of individuals is kept simple, reasoning activity is low when the historical is not active. Otherwise, the historical creates many individuals underneath and makes the reasoner slower. When reasoning is not active the historical has almost no penalty on the results. As expected, producers produce a much higher penalty than consumers, since they are the one which really produce the information. So, the higher number of producers in the experiments, the higher amount of information sent to OCP Server. The difference of time between a low number of consumers and a greater number has a lot to do with using the same machine for both clients and server.

Using Test 1.1 and Test 1.5 we can compare the cases where there is no reasoning and historical with that of the most powerful reasoner with historical. As we can see Test 1.1 scales extremely well, whereas Test 1.5 has some worse behaviour when it comes to scale. Despite this was expected, it confirms that huge ontologies are still not suitable for being exploited using standard reasoning mechanisms. The historical mode only makes the ontology grow quicker. Therefore, we can see the exponential curve coming sooner. However, Test 1.5 shows that most not intensive applications can perform very well using both historical and full reasoning setup. For intensive applications we can still use other configurations that scale better.

It must be noticed that 500 producers and consumers means 250.000 updates in OCP, and the total OCP processing time even with reasoning active is below 50 milliseconds. First producer - last consumer time also keeps low, around one second when context history is not active but it is reasoning.

Test 2 (see Figure 3). Number of operations per producer. In this experiment we only change the number of operations done in every experiment. This will show us how does the system evolve with time. When the historical is turned on, new entities will be created in every operation, so we will be able to see how does the system perform in these situations. We also try different configurations:

- Test 2.1. No reasoner. No rules. Historical off.
- Test 2.2. Pellet reasoner. Rules activated. Historical off.
- Test 2.3. Jena-full reasoner. Rules activated. Historical off.
- Test 2.4. No reasoner. No rules. Historical on.
- Test 2.5. Pellet reasoner. Rules activated. Historical on.
- Test 2.6. Jena-full reasoner. Rules activated. Historical on.

In these experiments we can see a few different things. On the one hand, is the reasoner is not active, the historical has almost no impact over time (not shown in Figure, but similar to Test 2.1). This is because the system does not need to traverse the ontology, but just set and get the items. When the reasoner is active, the historical means that the ontology makes bigger as time goes by and new information is introduced in the system. Therefore, reasoner will take longer, and the system will decrease it performance with time. In test

2.1 the time seems to decrease, rather than increase, but it just adjusts to its level, since in the first executions data must get into memory.

Regarding reasoning, we can see that, despite using the most expressive reasoning mode available, it produces very short opertation time results.

We have used the most aggressive historical mode, that stores everything, to test how does it perform. However, other historical modes have almost no impact over time.

There are some results that are not shown in the Figures. Internal OCP reasoning is virtually zero in almost all cases. It is below one millisecond. Since tests are run in the same machine, network time is also very close to zero. Isolated reasoning times and total OCP processing times are always below 50 ms (up to 29 ms. with 500 producers and consumers and reasoning activated).

6 Conclusions

In this paper we have presented OCP2 middleware for context information management and how does it perform in extreme situations (thousands of events per second). It shows remarkable results in the experiments. We have also seen that reasoning makes the system slower when it becomes bigger or when context history is active, despite they are still quite low. Depending on the use we will only need basic reasoning to use with rules, rather than OWL-DL or similar, and this can be configured within OCP.

Documentation, source code and latest OCP version is available at the web [3]. OCP has been used in various research projects and is currently deployed in several real systems in operation, among which we highlight the European project POPEYE (IST-2006-034241), CARDEA (FIT-350101-2007-14) and CARDINEA (TSI-020302-2009-43), so we can ensure that systems context-aware are no longer merely a research tool.

7 Future Work

As we have seen, historical hits OCP2 performance over time. There are three different modes of historical that minimize this effect, however, if we really need context history and performance, we need to separate the context history ontology from the main ontology. This has some problems related with temporal reasoning, but we are working to overpass them.

There are also some new features in which we are already working which offer us a way to create more advanced services. Temporal reasoning is still in an early stage. We are working on easier ways to express it. Another interesting task is the total distribution on different servers. Using a fully distributed structure would not only store just certain parts of the system and use small storage devices, but also we can use small storage devices and release the CPU load when reasoning, being able to use just lightweight devices. In addition, it would provide a more robust system in ubiquitous environments where the network may not always be available.

[3] http://darwin.inf.um.es/ocp/dokuwiki/

References

[1] T. Alsinet, C.I. Chesñevar, L. Godo, S. Sandri, and G. Simari. Formalizing argumentative reasoning in a possibilistic logic programming setting with fuzzy unification. *International Journal of Approximate Reasoning*, 48(3):711–729, 2008.

[2] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, and D. Riboni. A survey of context modelling and reasoning techniques. *Pervasive and Mobile Computing*, 2009.

[3] J.J. Carroll and I. Dickinson. Jena: implementing the semantic web recommendations. In *Proceedings of the 13th international World Wide Web conference*, pages 74–83. ACM New York, NY, USA, 2004.

[4] Sharma Chakravarthy, V. Krishnaprasad, Eman Anwar, and S.-K. Kim. Composite events for active databases: Semantics, contexts and detection. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 606–617. Morgan Kaufmann, 1994.

[5] Daniel D. Corkill. Collaborating software: Blackboard and multi-agent systems at the future. In *Proceedings of the International Lisp Conference*, New York, October 2003.

[6] G. De Giacomo and M. Lenzerini. TBox and ABox reasoning in expressive description logics. In *Principles of Knowledge representation and reasoning*, pages 316–327. Morgan Kaufmann, 1996.

[7] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[8] J. Hong, E. Suh, and S.J. Kim. Context-aware systems: A literature review and classification. *Expert Systems with Applications*, 36(4):8509–8522.

[9] Y. Ma, D.V. Kalashnikov, and S. Mehrotra. Toward managing uncertain spatial information for situational awareness applications. *IEEE Transactions on Knowledge and Data Engineering*, pages 1408–1423, 2008.

[10] I. Nieto, J.A. Botía, and A.F. Gómez-Skarmeta. Information and hybrid architecture model of the OCP contextual information management system. *Journal of Universal Computer Science*, 12(3):357–366, 2006.

[11] OMG. Data distribution service for real-time systems version 1.2. Technical report, Object Management Group, 2007.

[12] G. Papamarkos, A. Poulovassilis, and P. Wood. Event-condition-action rule languages for the semantic web. In *Workshop on Semantic Web and Databases, at VLDB'03*, 2003.

[13] D. Salber, A.K. Dey, and G.D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, page 441. ACM, 1999.

[14] Linnhoff-Popien C. Strang, T. A context modeling survey. In *Workshop on Advanced Context Modelling, Reasoning and Management UbiComp*. Citeseer, 2004.

[15] X.H. Wang, D.Q. Zhang, T. Gu, and H.K. Pung. Ontology based context modeling and reasoning using OWL. In *Proceedings of the second IEEE annual conference on pervasive computing and communications workshops*, volume 18. IEEE Computer Society Washington, DC, USA, 2004.

[16] D. Zhang, J. Cao, and J. Zhou. Extended Dempster-Shafer Theory in Context Reasoning for Ubiquitous Computing Environments. In *Proceedings of the 7th IEEE International Conference on Embedded and Ubiquitous Computing*, 2009.