P-Bench: benchmarking in data-centric pervasive application development

Sabina Surdu, Yann Gripay, Vasile-Marian Scuturici, Jean-Marc Petit

▶ **To cite this version:**

Sabina Surdu, Yann Gripay, Vasile-Marian Scuturici, Jean-Marc Petit. P-Bench: benchmarking in data-centric pervasive application development. Lecture Notes in Computer Science, 2013, Transactions on Large-Scale Data- and Knowledge-Centered Systems XI, 8290, pp.51-75. 10.1007/978-3-642-45269-7_3 . hal-01339246

HAL Id: hal-01339246
https://hal.science/hal-01339246

Submitted on 20 Mar 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

P-Bench: Benchmarking in Data-Centric Pervasive Application Development

Sabina Surdu[1,2,3], Yann Gripay[1,2], Vasile-Marian Scuturici[1,2], and Jean-Marc Petit[1,2]

[1] Université de Lyon, CNRS
[2] INSA-Lyon, LIRIS, UMR5205, F-69621, France
[3] UBB Cluj-Napoca, Faculty of Mathematics and Computer Science, RO-400084, Romania
{sabina.surdu,yann.gripay,vasile-marian.scuturici,jean-marc.petit}@insa-lyon.fr

Abstract. Developing complex data-centric applications, which manage intricate interactions between distributed and heterogeneous entities from pervasive environments, is a tedious task. In this paper we pursue the difficult objective of assessing the "easiness" of data-centric development in pervasive environments, which turns out to be much more challenging than simply measuring execution times in performance analyses and requires highly qualified programmers. We introduce P-Bench, a benchmark that comparatively evaluates the easiness of development using three types of systems: (1) the Microsoft StreamInsight unmodified Data Stream Management System, LINQ and C#, (2) the StreamInsight++ ad hoc framework, an enriched version of StreamInsight, that meets pervasive application requirements, and (3) our SoCQ system, designed for managing data, streams and services in a unified manner. We define five tasks that we implement in the analyzed systems, based on core needs for pervasive application development. To evaluate the tasks' implementations, we introduce a set of metrics and provide the experimental results. Our study allows differentiating between the proposed types of systems based on their strengths and weaknesses when building pervasive applications.

Keywords: pervasive environments, data-centric pervasive applications, heterogeneous data, continuous queries, benchmarking

1 Introduction

Nowadays we are witnessing the commencement of a new information era. The Internet as we know it today is rapidly advancing towards a worldwide Internet of Things [15], a planetary web that interconnects not only data and people, but also inanimate devices. Due to technological advances, we can activate the world of things surrounding us by enabling distributed devices to talk to one another, to signal their presence to users and to provide them with various data and functionalities.

In [16], Mark Weiser envisioned a world where computers vanish in the background, fitting smoothly into the environment and gracefully providing information and services to users, rather than forcing them to adapt to the intricate ambiance from the computing realm. Computing environments that arise in this context are generally referred to as *pervasive environments*, and applications developed for these environments are called *pervasive applications*. To achieve *easy to use* pervasive applications in a productive way, we must accomplish the realization of *easy to develop* applications.

Developing complex data-centric applications, which manage intricate interactions between distributed and heterogeneous entities from pervasive environments, is a tedious task, which often requires technical areas of expertise spanning multiple fields. Current implementations, which use DBMSs, Data Stream Management Systems (DSMSs) or just ad hoc programming (e.g., using Java, C#, .NET, JMX, UPnP, etc), cannot easily manage pervasive environments. Recently emerged systems, like Aorta [17], Active XML [1] or SoCQ [9], aim at easing the development of data-centric applications for pervasive environments. We call such systems *Pervasive Environment Management Systems* (PEMSs).

In this paper we pursue the difficult objective of assessing the "easiness" of data-centric development in pervasive environments, which turns out to be much more challenging than simply measuring execution times in performance analyses and requires highly qualified programmers. The main challenge lies in how to measure the easiness of pervasive application development and what metrics to choose for this purpose. We introduce Pervasive-Bench (P-Bench), a benchmark that comparatively evaluates the easiness of development using three types of systems: (1) the Microsoft StreamInsight unmodified DSMS [10], LINQ and C#, (2) the StreamInsight++ ad hoc framework, an enriched version of StreamInsight, which meets pervasive application requirements, and (3) our SoCQ PEMS [9], designed for data-centric pervasive application development. We define five tasks that we implement in the analyzed systems, based on core needs for pervasive application development. At this stage, we focus our study on applications built by a single developer. To evaluate the tasks' implementations and define the notion of *easiness*, we introduce a set of metrics. P-Bench allows differentiating between the proposed types of systems based on their strengths and weaknesses when building pervasive applications.

P-Bench is driven by our experience in building pervasive applications with the SoCQ system [9]. It also substantially expands our efforts to develop the ColisTrack testbed for SoCQ, which materialized in [8]. Nevertheless, the benchmark can evaluate systems other than SoCQ, being in no way limited by this PEMS.

We present a motivating scenario, in which we monitor medical containers transporting fragile biological content between hospitals, laboratories and other places of interest. A pervasive application developed for this scenario handles slower-changing data, similar to those found in classical databases, and distributed entities, represented as *data services*, that provide access to potentially unending dynamic data streams and to functionalities. Under reasonable as-

sumptions drawn by these types of scenarios, where we monitor data services that provide streams and functionalities, P-Bench has been devised to be a comprehensive benchmark.

To the best of our knowledge, this is the first study in the database community that addresses the problem of evaluating easiness in data-centric pervasive application development. Related benchmarks, like TPC variants [20] or Linear Road [3], focus on performance and scalability. While also examining these aspects, P-Bench primarily focuses on evaluating how easy it is to code an application, including deployment and evolution as well. This is clearly a daunting process, much more challenging than classical performance evaluation. We currently focus on pervasive applications that don't handle *big data*, in the order of petabytes or exabytes, e.g., home monitoring applications in intelligent buildings or container tracking applications. We believe the scope of such applications is broad enough to allow us to focus on them, independently of scalability issues. We strive to fulfill Jim Gray's criteria [7] that must be met by a domain-specific benchmark, i.e., relevance, portability, and simplicity. Another innovative feature of P-Bench is the inclusion of services, as dynamic, distributed and queryable data sources, which dynamically produce data, accessed through stream subscriptions and method invocations. In P-Bench, services become first-class citizens. We are not aware of similar works in this field. Another contribution of this paper is the integration of a commercial DSMS with service discovery and querying capabilities in a framework that can manage a pervasive environment.

This paper is organized as follows. Section 2 provides an insight into the requirements of data-centric pervasive application development, highlighting exigencies met by DSMSs, ad hoc programming and PEMSs. In Section 3 we describe the motivating scenario and we define the tasks and metrics from the benchmark. Section 4 presents the systems we assess in the benchmark, focusing on specific functionalities. In Section 5 we provide the results of our experimental study. Section 6 discusses the experimental results, highlighting the benefits and limitations of our implementations. Section 7 concludes this paper and presents future research directions.

2 Overview of Data-Centric Pervasive Applications

Pervasive applications handle data and dynamic data services[4] distributed over networks of various sizes. Services provide various *resources*, like streams and functionalities, and possibly static data as well. The main difficulties are to seamlessly integrate heterogeneous, distributed entities in a unified model and to homogeneously express the continuous interactions between them via declarative queries. Such requirements are met, to different extents, by pervasive applications, depending on the implementation.

DSMSs. DSMSs usually provide a homogeneous way to view and query relational data and streams, e.g., STREAM [2]. Some of them provide the ability to

[4] We will refer to a *data service* as a *service* or *data source* in the rest of the paper.

handle large-scale data, like large-scale RSS feeds in the case of RoSeS [4], or the ability to write SQL-like continuous queries. Nevertheless, developing pervasive applications using only DSMSs introduces significant limitations, highlighted by P-Bench.

Ad hoc programming using DSMSs. Ad hoc solutions, which combine imperative languages, declarative query languages and network protocols, aim at handling complex interactions between distributed services. Although they lead to the desired result, they are not long-term solutions, as P-Bench will show.

PEMSs. These systems aim at reconciling heterogeneous resources, like slower-changing data, streams and functionalities exposed by services in a unified representation in the query engine. PEMSs can be realized with many systems or approaches, such as Aorta [17], Active XML [1], SoCQ [9] or HYPATIA [5], to mention a few.

3 P-Bench

The P-Bench benchmark aims at providing an evaluation of different approaches to building data-centric pervasive applications. The common objective of benchmarks is to provide some way to evaluate which system yields better performance indicators when implementing an application, so that a "better" system can be chosen for the implementation [12]. Although we also consider performance in P-Bench, our focus is set on evaluating the easiness of data-centric pervasive application development with different types of systems: a DSMS, ad hoc programming and a PEMS.

To highlight the advantages of declarative programming, we ask that the evaluated systems implement tasks based on declarative queries. Some implementations will also require imperative code, others will not. We argue that one dimension of investigation when assessing the easiness of building pervasive applications is imperative versus declarative programming. A pervasive application seen as a declarative query over a set of services from the environment provides a logical view over those services, abstracting physical access issues. It also enables optimization techniques and doesn't require code compilation. When imperative code is included, restarting the system to change a query, i.e., recompiling the code, is considered as an impediment for the application developer.

3.1 Scenario and Framework

In our scenario, fragile biological matter is transported in sensor-enhanced medical containers, by different transporters. During the containers' transportation, temperature, acceleration, GPS location and time must be observed. Corresponding sensors are embedded in the container: a temperature sensor to verify temperature variations, an accelerometer to detect high acceleration or deceleration, a timer to control the deadline beyond which the transportation is unnecessary and a GPS to know the container position at any time.

A supervisor determines thresholds for the different quality criteria a container must meet, e.g., some organic cells cannot be exposed to more than 37° C. When a threshold is exceeded, the container sends a text message (e.g., via SMS) to its supervisor.

In our scenario, only part of these data are static and can be stored in classical databases: the medical containers' descriptions, the different thresholds. All the other data are dynamically produced by distributed services and accessed through method invocations (e.g., get current location) or stream subscriptions (e.g., temperature notifications). Moreover, services can provide additional functionalities that change the environment, like sending some messages (e.g., by SMS) when an alert is triggered; they can also provide access to data stored in relations, if necessary. Therefore, our scenario is representative for pervasive environments, where services provide static data, dynamic data streams and methods that can be invoked [9]. It is also compatible with existing scenarios for DSMSs (like the one from Linear Road), but services are promoted as first-class citizens.

A data service that models a device in the environment has an URL and accepts a set of operations via HTTP. P-Bench contains CAR, MEDICAL CONTAINER and ALERT services, which expose streams of car locations, of medical container temperature notifications, the ability to send alert messages when exceptional situations occur, etc.

We developed a framework to implement this scenario (Figure 1). Since we use a REST/HTTP-based protocol to communicate with services, they can be integrated independently of the operating system and programming language. Moreover, assessed systems can be equipped with modules for dynamic service discovery.

The World Simulator Engine is a C# application that runs on a Windows 2008 Server machine and simulates (i.e., generates) services in the environment. The simulator accepts different options, like the number of cars, the places they visit, the generation of medical containers, etc. Services' data rate is also parameterizable (e.g., how often a car emits its location). The engine uses the Google Maps Directions API Web Service to compute real routes of cars.

The Control & Visualization Interface allows visualizing services in the World, and writing and sending declarative queries to a query engine. The Visualization Interface runs on an Apache Web server; the server side is developed in PHP. On the client side, the web user interface is based on the Google Maps API to visualize the simulated world on a map, and uses Ajax XML HTTP Request to load the simulated state from the server side. Several remote clients can connect simultaneously to the same simulated environment, by using their Web browser. This user interface is not mandatory for our benchmark, but it does provide a nice way of visualizing services and the data they supply. Declarative queries can be written using an interface implemented as an ASP.NET Web application that runs on the Internet Information Services server. We thoroughly describe this scenario and framework in our previous paper on ColisTrack [8].

In our experiments we eliminate the overhead introduced by our web interface. In the StreamInsight and StreamInsight++ implementations we use an in-process server and send queries from the C# application that interacts with the server. In the case of SoCQ, we write and send queries from SoCQ's interface.

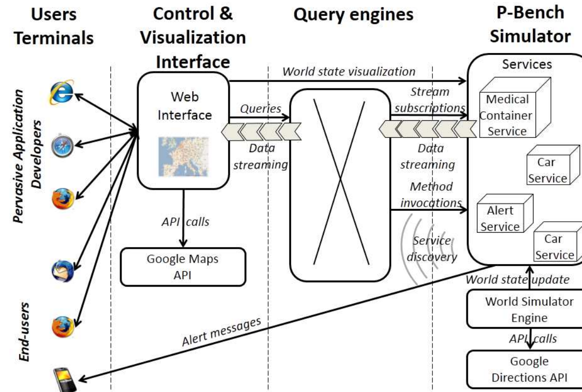

Fig. 1. Scenario framework architecture

3.2 Benchmark Tasks

We define five benchmark tasks to evaluate the implementation of our scenario with the assessed systems. The main challenge in pervasive applications is to homogeneously express interactions between resources provided by dynamically discovered services, e.g., data streams, methods and static data. Therefore, we wrap tasks' definitions around functionalities dictated by these necessities. Each task is built around a main functionality that has to be implemented by a system in order to fulfill the task's objective. The parameters of the tasks are services specific to our scenario. These parameters can easily be changed, so that a task can be reformulated on any pervasive environment-based scenario, whilst maintaining its specified objective. The difficulty of the tasks is incremental. We start with a task that queries a single data stream from a given service, and we end with a task that combines heterogeneous resources from dynamically discovered services of different types.

Since P-Bench is concerned with assessing development in pervasive environments, our tasks are defined in the scope of pervasive applications. Other types of applications like data analysis applications are not in the focus of our current study.

Task 0: Startup. The objective of this task is to prepare the assessed systems for the implementation of the scenario. It includes the system-specific description of the scenario, i.e., data schema, additional application code, etc.

Task 1: Localized, single stream supervision. The objective of this task is to monitor a data stream provided by a service that had been localized in advance, i.e., dynamic service discovery is not required. Task 1 tracks a single moving car and uses a CAR service URL. The user is provided, at any given time instant, with the last reported location of the monitored car.

Task 2: Multiple streams supervision. The objective of this task is to monitor multiple data streams provided by dynamically discovered services. Task 2 tracks all the moving cars. The user is provided, at any given time instant, with the last reported location of each car.

Task 3: Method invocation. The objective of this task is to invoke a method provided by a dynamically discovered service. Task 3 provides the user with the current location of a medical container, given its identifier.

Task 4: Composite data supervision. The objective of this task is to combine static data, and method invocations and data streams provided by dynamically discovered services, in a monitoring activity. Task 4 monitors the temperatures of medical containers and sends alert messages when the supervised medical containers exceed established temperature thresholds.

3.3 Benchmark Metrics

Similarly to the approach from [6], we identify a set of pervasive application quality assurance goals: easy development, easy deployment and easy evolution. Since easiness alone cannot be a sole criterion for choosing a system, we also introduce the performance goal to assess the efficiency of a system under realistic workloads. Based on these objectives we define a set of metrics that we think fits best for evaluating the process of building pervasive applications.

We define the life cycle of a task as the set of four stages that must be covered for its accomplishment. Each stage is assessed through related metrics and corresponds to one of the quality assurance goals:

- development - metrics from this stage assess the easiness of task development;
- deployment - metrics from this stage evaluate the easiness of task deployment;
- performance - in this stage we assess system performance, under realistic workloads;
- evolution - metrics from this stage estimate the impact of the task evolution, i.e., how easy it is to change the current implementation of the task, so that it adapts to new requirements. The task's objective remains unmodified.

By defining the life cycle of a task in this manner, we adhere to the goal of agility [13] in P-Bench. Agility spans three life cycle stages: development, deployment and evolution. Since we are not concerned with big data, we don't focus on scale agility.

We now define a set of metrics for each of the four stages:

Development. We separate task development on two levels: imperative code (written in an imperative programming language, e.g., C#) and declarative code

(written in a declarative query language, e.g., Transact-SQL). We measure the easiness and speed in the development of a task through the following metrics:

- LinesOfImperativeCode outputs the number of lines of imperative code required to implement the task (e.g., code written in Java, C#). The tool used to assess this metric is SLOCCount [21]. We evaluate the middleware used to communicate with services in the environment, but we exclude predefined class libraries from our assessment (e.g., classes from the .NET Base Class Library);
- NoOfDeclarativeElements provides the number of declarative elements in the implementation of the task. We normalize a query written in a declarative language in the following manner. We consider a set of language-specific declarative keywords describing query clauses, for each of the evaluated systems. The number of declarative elements in a query is given by the number of keywords it contains (e.g., a SELECT FROM WHERE query in Transact-SQL contains three declarative elements);
- NoOfQueries outputs the number of declarative queries required for the implementation of the task;
- NoOfLanguages gives the number of imperative and declarative languages that are used in the implementation of the task;
- DevelopmentTime roughly estimates the number of hours spent to implement the task, including developer training time and task testing, but excluding the time required to implement the query engine or the middleware used by the systems to interact with services.

Deployment. The deployment stage includes metrics:

- NoOfServers gives the number of servers required for the task (e.g., the StreamInsight Server);
- NoOfSystemDependencies outputs the number of system-specific dependencies that must be installed for the task;
- IsOSIndependent indicates whether the task can be deployed on any operating system (e.g., Windows, Linux, etc).

Performance. Once we implemented and deployed the task, we can measure the performance of this implementation. We need now to rigorously define accuracy and latency requirements.

The accuracy requirement states that queries must output correct results. Our work for an accuracy checking framework in a pervasive environment setting is ongoing. Using this framework we will compute the correct results for queries in a given task, we will calculate the results obtained when implementing the task with an assessed system, and finally, we will characterize the accuracy of the latter results using Precision and Recall metrics. We will consider both the results of queries and the effects that query executions have on the environment.

We place an average latency requirement of 5 seconds on continuous queries, i.e., on average, up to 5 seconds can pass between the moment an item (i.e., a tuple or an event) is fed into a query and the moment the query outputs a

result based on this item. We set a query execution time of 60 seconds. When assessing performance for systems that implement dynamic service discovery, a query starts only after all the required services have been discovered, but during query execution both StreamInsight++ and SoCQ continue to process messages from services that appear and disappear on and from the network.

To evaluate performance, we consider the average latency and accuracy requirements described above and define a set of metrics for continuous queries. In the current implementation, the metrics are evaluated by taking into account the average latency requirement, but our accuracy checking framework will allow us to evaluate them with respect to the accuracy constraints as well. The performance stage metrics are:

– MaxNoDataSources gives the maximum number of data sources (i.e., services) that can feed one continuous query, whilst meeting accuracy and latency requirements. We assign a constant data rate of 10 events/minute for each data source;
– MaxDataRate outputs the maximum data rate for the data sources that feed a continuous query, under specified accuracy and latency requirements. All the sources are supposed to have the same constant data rate. This metric is expressed as number of events per second. We are not interested in extremely high data rates for incoming data, so we will evaluate the task up to a data rate of 10.000 events/second. Unless specified otherwise in the task, this metric is evaluated for 10 data sources;
– NoOfEvents is the number of processed events during query execution when assessing the MaxDataRate metric. This metric describes the limitations of our implementations and hardware settings, more than system performance;
– AvgLatency outputs the average latency for a continuous query, given a constant data rate of 10 events/second for the data sources that feed the query. AvgLatency is expressed in milliseconds and is computed across all the data sources (10 by default) that feed a continuous query, under specified accuracy requirements.

Evolution. The evolution stage encompasses metrics that quantify the impact that new requirements or changes have on the whole task. The evolution of a task does not suffer radical changes (i.e., we don't modify a task that subscribes to a stream, to invoke a method in its updated version). A task's parameters, e.g., the services, may change, but the specified objective for a task is maintained. This stage contains the following metrics:

– ChangedImperativeCode outputs the number of lines of imperative code that need to be changed (added, modified or removed), when the task evolves, in order to accomplish newly specified requirements. Lines of imperative code are counted like in the case of the LinesOfImperativeCode metric;
– ChangedDeclarativeElements provides the number of declarative elements that need to be changed in any way (added, modified or removed), in order to update the task. Counting declarative elements is performed like in the case of the NoOfDeclarativeElements metric.

Metrics in this stage provide a description of the reusability dimension when developing pervasive applications. We are assessing the energy and effort devoted to the process of task evolution.

4 Assessed Systems

The DSMS we use in P-Bench is StreamInsight. To accomplish the tasks in an ad hoc manner, we enrich StreamInsight with dynamic service discovery features, obtaining a new framework: StreamInsight++. As a PEMS, we use SoCQ. To communicate with services in the environment, we use UbiWare, the middleware we developed in [14] to facilitate application development for ambient intelligence.

StreamInsight was chosen based on the high familiarity with the Microsoft .NET-based technologies. We chose SoCQ because of the expertise our team has with this system and the ColisTrack testbed. We don't aim at conducting a comprehensive study of DSMSs or PEMSs, but P-Bench can as well be implemented in other DSMSs like [18], [4], [2], or PEMSs like [1] or [5].

4.1 Microsoft StreamInsight

Microsoft StreamInsight [10] is a platform for the development and deployment of Complex Event Processing (CEP) applications. It enables data stream processing using the .NET Framework. For pervasive application development, additional work has to be done in crucial areas, like service discovery and querying. To execute queries on the StreamInsight Server, one requires a C# application to communicate with the server. We enrich this application with a Service Manager module, which handles the interaction with the services in the environment and which is based on UbiWare.

As described in the technical documentation [19], StreamInsight processes event streams coming in from multiple sources, by executing continuous queries on them. Continuous queries are written in Language-Integrated Query (LINQ) [11]. StreamInsight's run-time component is the StreamInsight server, with its core engine and the adapter framework. Input adapters read data from event sources and deliver them to continuous queries on the server, in a push manner. Queries output results which flow, using pull mechanisms, through output adapters, in order to reach data consumers.

Figure 2 shows the architecture of an application implemented with StreamInsight (similar to [19]). Events flow from network sources in the pervasive environment through input adapters into the StreamInsight engine. Here they are processed by continuous queries, called *standing queries*. For simplicity, we depict data streaming in from one CAR service and feeding one continuous query on the server. The results are streamed through an output adapter to a consumer application. Static reference data (e.g., in-memory stored collections or SQL Server data) can be included in the LINQ standing queries specification.

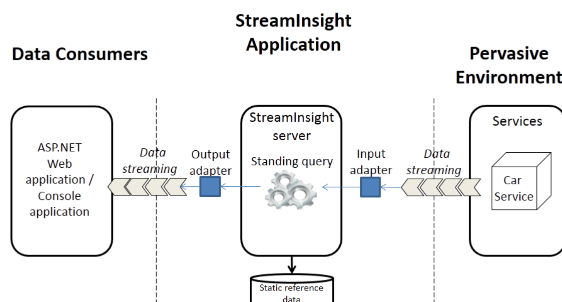

Fig. 2. StreamInsight application architecture

4.2 StreamInsight++

StreamInsight contains a closed source temporal query engine that cannot be changed. Instead, we enrich the Service Manager with dynamic service discovery capabilities, using ad hoc programming, thus obtaining the StreamInsight++.

The enriched Service Manager allows the user of StreamInsight++ to write queries against dynamically discovered services. It can be thought of as the middleware between the system and the services in the environment, or the *service wrapper* that allows both service discovery and querying. The service access mechanism uses the REST/HTTP-based protocol mentioned in Section 3. The Service Manager delivers data from discovered services to input adapters.

4.3 SoCQ

We designed and implemented the Service-oriented Continuous Query (SoCQ) engine [9], a PEMS that enables the development of complex applications for pervasive environments using declarative service-oriented continuous queries. These SQL-like queries combine conventional and non-conventional data, namely slower-changing data, dynamic streams and functionalities, provided by services.

Within our data-oriented approach, we built a complete data model, namely the SoCQ data model, which enables services to be modeled in a unified manner. It also provides a declarative query language to homogeneously handle data, streams and functionalities: Serena SQL. In a similar way to databases, we defined the notion of *relational pervasive environment*, composed of several *eXtended Dynamic Relations*, or XD-Relations. The schema of an XD-Relation is composed of real and/or virtual attributes [9]. Virtual attributes represent parameters of various methods, streams, etc, and may receive values through query operators. The schema of an XD-Relation is further associated with binding patterns, representing method invocations or stream subscriptions.

SoCQ includes service discovery capabilities in the query engine. The service discovery operator builds XD-Relations that represent sets of available services providing required data. For example, an XD-Relation CAR could be the result of

such an operator, and be continuously updated when new CAR services become available and when previously discovered services become unavailable.

5 Benchmark Experiments

In this section we present the comparative evaluation of the chosen systems. For each task, we will describe its life cycle on StreamInsight, StreamInsight++ and SoCQ. We start with the development stage, continue with deployment and performance and end with task evolution. We rigorously assess each task through the set of metrics we previously defined. At the end of each subsection dedicated to a task we provide a table with metrics results and a short discussion. The experiments were conducted on a Windows Server 2008 machine, with a 2.67GHz Intel Xeon X5650 CPU (4 processors) and 16 GB RAM.

5.1 Assessing Performance

We present our system-specific evaluation approach for the performance stage:

StreamInsight and StreamInsight++. In this case we use an in-process server. We connect to one or more service streams and deliver incoming data to an input adapter. We assess the time right before the input adapter enqueues an event on the server and the time right after the output adapter dequeues the event from the server. The time interval delimited by the enqueue and dequeue moments represents the event's latency. Average latency is computed incrementally based on individual event latencies. We also enqueue CTI events on the server, i.e., special events specific to StreamInsight, used to advance application time, but we compute average latency by taking into account only events received from environment services.

By evaluating latency in this manner, we assess the performance of the StreamInsight engine together with the adapter framework and middleware that we implemented, and not the pure performance of the StreamInsight engine.

SoCQ. The average latency is computed by comparing events from streams of data services, to events from the query output stream. An event from a service is uniquely identified by the service URL and the service-generated event timestamp. A unique corresponding event is then expected from the query output stream. A latency measurement tool has been developed to support the latency computation, based on UbiWare: it launches the task query in the query engine, connects to the query result output stream, connects to a number of services, and then matches expected events from services and query output events from the query engine. The difference between the arrival time of corresponding events at the measurement tool provides a latency for each expected event.

5.2 Task 0: Startup

The objective of this task is to prepare the evaluated systems for the implementation of Tasks 1 to 4. The latter can be implemented independently from one

another, but they all require the prior accomplishment of the Startup task. We describe the schema of our scenario in system-specific terms. We also present any additional modules that need to be implemented. Task 0 uses the UbiWare middleware [14] previously mentioned, to interact with services in the environment. UbiWare uses a REST/HTTP-based protocol for this purpose.

The developer that implemented the Startup task in StreamInsight and StreamInsight++ has a confident level of C#, .NET and LINQ, but has never developed applications for StreamInsight before. We don't embark on an incremental development task, evolving from StreamInsight to StreamInsight++. We consider them to be independent, separate systems, hence any common features are measured in the corresponding metrics, for each system.

The same developer also accomplished the Startup task in SoCQ, without having any prior knowledge about the system and the SQL-like language it provides.

Development. *StreamInsight and StreamInsight++.* We implement C# solutions that handle the interaction with the StreamInsight server. They contain entities specific to StreamInsight (input and output configuration classes and adapters, etc) and entities that model data provided by services in P-Bench (CAR LOCATION, TEMPERATURE NOTIFICATION classes, etc). To interact with environment services, these implementations also use and enrich the Service Manager specific to StreamInsight or StreamInsight++.

StreamInsight. Task 1 is the only task that can be fully implemented with StreamInsight, as it doesn't require service discovery (the URL of the car service that represents the car to be monitored is provided). Therefore, we implement a C# solution, which handles the interaction with the StreamInsight server, to prepare the system for Task 1. The solution contains the following entities:

- a CAR LOCATION class, that models location data provided by a CAR service (latitude, longitude, timestamp and car id);
- a CAR DATA SOURCE module, that is part of the Service Manager, and delivers incoming car locations (from the given service URL) to an input adapter;
- input and output configuration classes, to specify particulars of data sources and consumers;
- input and output adapter factory classes, responsible for creating input and output adapters;
- a typed input adapter, which receives a specific CAR LOCATION event from the CAR DATA SOURCE in a push manner and enqueues this event, using push mechanisms, into the StreamInsight server;
- an output adapter, which dequeues results from the query on the StreamInsight server;
- an additional BENCHMARK TOOLS class, that manages application state, computes latency, etc.

StreamInsight++. StreamInsight++ can implement all the tasks. The C# solution we built to enrich StreamInsight and to communicate with the StreamInsight server is much more complex than the one used with raw StreamInsight, but there are some common features. The solution contains the following entities:

- apart from the CAR LOCATION class, we developed C# classes that model medical containers and their temperature notifications, i.e., MEDICAL CONTAINER and TEMPERATURE NOTIFICATION;
- we enriched the CAR DATA SOURCE module to encompass dynamic discovery, so as to deliver car locations from dynamically discovered cars;
- we added medical containers data source modules specific to Tasks 3 and 4, respectively, i.e., MEDICAL CONTAINER DATA SOURCE and TEMPERATURE NOTIFICATION DATA SOURCE;
- additional input adapter factory classes were developed, for the newly added input adapters (for medical containers and medical containers temperature notifications);
- classes that contain the input configuration, output configuration, output adapter factory and output adapter were maintained (we chose to implement an untyped output adapter);
- extra input adapters were developed, to handle the diversity of input events from the pervasive environment, i.e., medical containers dynamic discovery messages and medical containers temperature notifications;
- the BENCHMARK TOOLS class was extended to encompass methods specific to Tasks 3 and 4.

SoCQ. All the tasks can be implemented with SoCQ. SoCQ already contains the middleware required for the services in P-Bench, but to provide a fair comparison with the other systems, we will assess the code in SoCQ's middleware as well.

We provide a SoCQ schema of our scenario, written in Serena SQL. Listing 1 depicts the set of XD-Relations, which abstract the distributed entities in the pervasive environment. This is the only price the application developer needs to pay to easily develop data-centric pervasive applications with SoCQ: gain an understanding of SoCQ and Serena SQL and model the pervasive environment as a set of XD-Relations, yielding a relational pervasive environment. CAR, MEDICALCONTAINER and SUPERVISORMOBILE are finite XD-Relations, extended with virtual attributes and binding patterns in order to provide access to stream subscriptions and method invocations. SUPERVISE is a simple dynamic relation, with no binding patterns, yet all four relations are specified in a consistent, unified model, in the Serena SQL. On top of the relational pervasive environment, the developer can subsequently write applications as continuous queries, which reference data services from the distributed environment and produce data.

Deployment. *StreamInsight and StreamInsight++.* To attain this task with StreamInsight and StreamInsight++, one requires .NET, a C# compiler, SQL Server Compact Edition, a Windows operating system and the StreamInsight server. This minimum setting is necessary for Tasks 1 to 4, with some additional task-dependent prerequisites.

SoCQ. The deployment machine must have the SoCQ Server and a Java Virtual Machine. Any operating system can support this task.

Evolution. The Startup task prepares the system to handle a pervasive environment, based on entities from the scenario we proposed. If we change the

```
CREATE RELATION Car (carID STRING PRIMARY KEY, carService
    SERVICE, latitude STRING VIRTUAL,
    longitude STRING VIRTUAL, locDate DATE VIRTUAL)
  USING BINDING PATTERNS (
    locationNotification[carService] () : (latitude,
        longitude, locDate) STREAMING);

CREATE RELATION MedicalContainer (mcID STRING PRIMARY KEY,
    mcService SERVICE, temperatureDate DATE VIRTUAL,
    temperatureValue REAL VIRTUAL, locDate DATE VIRTUAL,
    latitude STRING VIRTUAL, longitude STRING VIRTUAL,
    timeDate DATE VIRTUAL, timeout REAL VIRTUAL)
  USING BINDING PATTERNS (
    temperatureNotification[mcService] () : (temperatureDate,
        temperatureValue) STREAMING,
    getTemperature[mcService] () : (temperatureDate,
        temperatureValue),
    getLocation[mcService] () : (locDate , latitude, longitude),
    getTimeout[mcService] () : (timeDate , timeout) );

CREATE RELATION SupervisorMobile (mobileID STRING PRIMARY KEY,
    phone STRING, alertService SERVICE, alertDate DATE VIRTUAL,
    alertMessage STRING VIRTUAL, alertSent BOOLEAN VIRTUAL)
  USING BINDING PATTERNS (
    sendSMS [alertService] (phone, alertDate, alertMessage) :
        (alertSent) );

CREATE RELATION Supervise (mobileID STRING, mcID STRING,
    temperatureThreshold REAL,
    fromLatitude STRING, fromLongitude STRING, fromDate DATE,
    toLatitude STRING, toLongitude STRING, toDate DATE,
    PRIMARY KEY (mobileID, mcID) );
```

Listing 1: SoCQ schema for the motivating scenario

pervasive environment, we must rebuild the Startup task in order to handle different services, which expose different types of data and / or use different service wrappers.

StreamInsight and StreamInsight++. We must redevelop the C# solutions if the service wrappers change. If the service access mechanisms don't change, the middleware can remain unmodified, i.e., ChangedImperativeCode won't consider the ∼3700 lines of code that compose the middleware implemented for StreamInsight and StreamInsight++. Other classes might be kept if some data provided by services from the initial environment are preserved.

SoCQ. In SoCQ, we need to build a different schema, in Serena SQL, for the new pervasive environment. If the service wrappers change, then the imperative code for the middleware must be reimplemented. If the middleware remains unmodified, no line of imperative code is impacted in the evolution stage, i.e., ChangedImperativeCode will be 0.

Task discussion. The time and effort devoted to self-training and implementing Task 0 are considerably higher in the StreamInsight-based implementations than in SoCQ (see Table 1). The former can only be deployed on Windows machines. SoCQ needs a smaller number of system dependencies and can be deployed on any operating system. If we switch to a different scenario, Task 0 needs to be reimplemented, which translates to a significant amount of changed lines of imperative code in all the systems, if the service access mechanisms change. If the middleware doesn't change, the exact amount of changed code depends on

the preservation of some services from the initial environment; in StreamInsight and StreamInsight++ we need to modify imperative code, whereas SoCQ requires changing only declarative elements. Table 1 shows figures for the worst-case situation, where all services and their access mechanisms are changed.

Table 1. Task 0 metrics

Stage	Metric	SI	SI++	SoCQ
Development	LinesOfImperativeCode	4323	5186	26500[5]
	NoOfDeclarativeElements	0	0	13
	NoOfQueries	0	0	4
	NoOfLanguages	1	1	2
	DevelopmentTime	120	160	16
Deployment	NoOfServers	1	1	1
	NoOfSystemDependencies	3	3	1
	IsOSIndependent	No	No	Yes
Evolution	ChangedImperativeCode	∼4323	∼5186	∼11000
	ChangedDeclarativeElements	0	0	∼13

5.3 Task 1: Localized, Single Stream Supervision

Task 1 tracks one moving car. Its input is a CAR service URL and a stream of locations from the monitored car. The output of the task is a stream that contains the LOCATIONTIMESTAMP, LATITUDE, LONGITUDE and CARID of the car, i.e., the user is provided with the car's stream of reported locations. The task's objective is to monitor a data stream provided by a CAR service that had been localized in advance, i.e., dynamic service discovery is not required.

Development. *StreamInsight and StreamInsight++*. We require one LINQ query in order to track a given car (Listing 2a). We need additional C# code to create a query template that represents the business logic executed on the server, instantiate adapters, bind a data source and a data consumer to the query, register the query on the server and start and stop the continuous query. Dynamic discovery is not required for this task.

SoCQ. In SoCQ, the developer writes a car tracking query in Serena SQL (Listing 2b). It subscribes to a stream of location data from the CAR XD-Relation, based on a CAR service URL. No imperative code is needed.

[5] The SoCQ engine source code contains about 26500 lines of Java code. It encompasses the UbiWare generic implementation (client-side and server-side, about 11000 lines of code), the core of the SoCQ engine (data management and query processing, about 13200 lines), and some interfaces to control and access the SoCQ engine (2 Swing GUI and a DataService Interface, about 2300 lines). For StreamInsight and StreamInsight++, LinesOfImperativeCode assesses only the task application and Service Manager code (we don't have access to StreamInsight's engine implementation).

```
FROM Car IN CarSupervision
SELECT Car.CarID, Car.Latitude, Car.Longitude,
    Car.LocationTimestamp;
```
(a) Car supervision query in LINQ

```
CREATE VIEW STREAM carSupervision (carID STRING, locDate
    DATE, locLatitude STRING, locLongitude STRING)
AS SELECT c.carID, c.locDate, c.latitude, c.longitude
STREAMING UPON insertion
FROM Car c
WHERE c.carService = "http://127.0.0.1:21000/Car"
USING c.locationNotification [1];
```
(b) Car supervision query in SoCQ's Serena SQL

Listing 2: Car supervision queries

Deployment. *StreamInsight, StreamInsight++ and SoCQ.* For this task, the same prerequisites as for Task 0 are required, for all the implementations.

Performance. *StreamInsight, StreamInsight++ and SoCQ.* For this task we assess metrics MaxDataRate, NoOfEvents and AvgLatency, since we track one car.

Evolution. *StreamInsight, StreamInsight++ and SoCQ.* The user may want to track a different car, which means changing the CAR service URL. In our StreamInsight-based approaches, this requires changing and recompiling the imperative code, to provide the new URL. The LINQ query remains unchanged. In SoCQ, we supply a different CAR service URL in the declarative query code.

Task discussion. The effort required to develop and update the task is more intense in the StreamInsight-based implementations, which can be deployed only on Windows machines and require 2 languages, LINQ and C#, and more dependencies (see Table 2). The SoCQ implementation uses 1 language (Serena SQL), needs 1 dependency and no imperative code, and can be deployed on any operating system, but it yields a higher average latency. All the systems achieved a MaxDataRate of 10.000 events/second under specified latency requirements.

5.4 Task 2: Multiple Streams Supervision

Task 2 tracks all the moving cars. The input of this task is represented by notification messages sent by services in the environment when they appear or disappear and by streams of interest emitted by services monitored in the task, i.e., car location streams from monitored cars. The output of this task is a stream that provides the LocationTimestamp, Latitude, Longitude and CarId of the monitored cars. The user is hence provided with the reported locations of each car. Task 2's objective is to monitor multiple data streams provided by dynamically discovered CAR services.

Development. *StreamInsight++.* This implementation is similar to the one from Task 1, but the CAR DATA SOURCE receives events from all the streams the application subscribed to. It delivers them in a push manner to the input adapter. Hence, the LINQ query for this task is identical to the one described in Listing 2a.

Table 2. Task 1 metrics

Stage	Metric	SI	SI++	SoCQ
Development	LinesOfImperativeCode	33	33	0
	NoOfDeclarativeElements	2	2	6
	NoOfQueries	1	1	1
	NoOfLanguages	2	2	1
	DevelopmentTime	4	4	1
Deployment	NoOfServers	1	1	1
	NoOfSystemDependencies	3	3	1
	IsOSIndependent	No	No	Yes
Performance	MaxDataRate	10000	10000	10000
	NoOfEvents	350652	350652	360261
	AvgLatency	0.5	0.5	1.34
Evolution	ChangedImperativeCode	1	1	0
	ChangedDeclarativeElements	0	0	1

SoCQ. The SoCQ implementation is similar to the one described for Task 1. The only requirement is to write the car tracking query. The query for this task is identical with the one depicted in Listing 2b, except it doesn't encompass a filter condition, since we are tracking all the cars.

Deployment. *StreamInsight++ and SoCQ.* The prerequisites for deployment are identical with those mentioned in Task 0.

Performance. *StreamInsight++ and SoCQ.* We evaluate all the metrics from the performance stage. We compute MaxDataRate, NoOfEvents and AvgLatency across events coming in from all data sources, for a constant number of 10 data sources.

Evolution. *StreamInsight++ and SoCQ.* A new requirement for this task can be to track a subgroup of moving cars. In StreamInsight++ we need to change the imperative code, to check the URL of the data source discovered by the system. In SoCQ, we need to add a filter predicate in the continuous query.

Task discussion. SoCQ provides a convenient approach to development, deployment and evolution, without imperative code, obtaining better results for metrics NoOfLanguages, NoOfSystemDependencies and IsOSIndependent (Table 3). StreamInsight++ achieves superior performance when assessing MaxNoDataSources and MaxDataRate. We believe this implementation could do better, but in our hardware setting we noticed a limit of 18.000 events that are received by the StreamInsight engine each second; hence this is not a limitation imposed by StreamInsight. For our scenario, the performance values obtained by SoCQ are very good as well. We have multiple threads in our StreamInsight++ application to subscribe to multiple streams, so the thread corresponding to the StreamInsight++ output adapter is competing with existing in-process threads. Therefore, the average latency we observe from the adapters is higher than the StreamInsight's engine pure latency and than the average latency measured for

SoCQ. This task cannot be implemented in StreamInsight, due to lack of dynamic service discovery.

Table 3. Task 2 metrics

Stage	Metric	SI	SI++	SoCQ
Development	LinesOfImperativeCode	NA	31	0
	NoOfDeclarativeElements	NA	2	5
	NoOfQueries	NA	1	1
	NoOfLanguages	NA	2	1
	DevelopmentTime	NA	4	1
Deployment	NoOfServers	NA	1	1
	NoOfSystemDependencies	NA	3	1
	IsOSIndependent	NA	No	Yes
Performance	MaxNoDataSources	NA	5000	2500
	MaxDataRate	NA	1700	750
	NoOfEvents	NA	976404	443391
	AvgLatency	NA	13.53	0.79
Evolution	ChangedImperativeCode	NA	1	0
	ChangedDeclarativeElements	NA	0	1

5.5 Task 3: Method Invocation

Task 3 provides the location of a medical container. The input of this task is represented by a medical container identifier and notification messages sent by services in the environment when they appear or disappear. Its output is the current location of the container, i.e., the LocationTimestamp, Latitude and Longitude. The objective of this task is to invoke a method provided by a dynamically discovered medical container service.

Development. *StreamInsight++*. We create a SQL Server database and dynamically update a table in the database with available medical container services. An input adapter delivers medical container services discovered by Service Manager to a simple LINQ continuous query, whose results are used to update the medical container services table in SQL Server. Based on the input container identifier (an mcID field), the application looks up the medical container URL in the SQL Server table. From imperative code, it calls the getLocation method exposed by the medical container service, which outputs the current location of the container.

SoCQ. We write a simple Serena one-shot query that uses the MedicalContainer XD-Relation, defined in the SoCQ schema (Listing 3). We manually submit this query using SoCQ's interface.

Deployment. *StreamInsight++*. Apart from the prerequisites described in Task 0, to implement Task 3 we also need an installed instance of SQL Server.

```
SELECT latitude, longitude, locDate
FROM MedicalContainer
WHERE mcID="12345"
USING getLocation;
```

Listing 3: Locating medical container query in SoCQ

SoCQ. Task 0 prerequisites hold for this task implemented in SoCQ.

Performance. *StreamInsight++ and SoCQ.* We don't assess performance metrics for this task, as it encompasses a one-shot query. Assessing service discovery performance is out of the scope of this evaluation.

Evolution. *StreamInsight++ and SoCQ.* The user may want to locate a different medical container. In StreamInsight++ we need to supply a different container identifier in the imperative application. In SoCQ we supply a different medical container identifier in the Serena query.

Task discussion. For this task as well development time and effort are minimal in the SoCQ implementation, which doesn't need imperative code (see Table 4). In StreamInsight++ we also need an additional instance of SQL Server. If SoCQ requires only Serena SQL, StreamInsight++ requires C#, LINQ and Transact-SQL (to interact with SQL Server).[6] Metrics NoOfSystemDependencies and IsOSIndependent yield better values for SoCQ. This task cannot be implemented in StreamInsight, because it requires dynamic service discovery.

Table 4. Task 3 metrics

Stage	Metric	SI	SI++	SoCQ
Development	LinesOfImperativeCode	NA	102	0
	NoOfDeclarativeElements	NA	11	4
	NoOfQueries	NA	4	1
	NoOfLanguages	NA	3	1
	DevelopmentTime	NA	8	1
Deployment	NoOfServers	NA	2	1
	NoOfSystemDependencies	NA	3	1
	IsOSIndependent	NA	No	Yes
Evolution	ChangedImperativeCode	NA	1	0
	ChangedDeclarativeElements	NA	0	1

5.6 Task 4: Composite Data Supervision

Task 4 monitors the temperatures of medical containers and sends alert messages when the supervised medical containers exceed established temperature thresholds. The input of this task is represented by notification messages sent

[6] We will replace Transact-SQL with LINQ to SQL.

by services from the environment when they appear or disappear, and by streams of temperature notifications from the medical containers of interest. The output of this task is a stream of calls to methods from ALERT services. Task 4's objective is to combine in a monitoring activity static data (temperature thresholds and supervision related data), and method invocations (send alert messages) and data streams (temperature notifications) provided by dynamically discovered ALERT and MEDICAL CONTAINER services.

Development. *StreamInsight++*. This implementation integrates the StreamInsight Server, as well as SQL Server, LINQ and C#. We need SQL Server to hold supervision related data (which supervisors monitor which medical containers) and dynamically discovered ALERT SERVICES. For the incoming medical containers temperature notifications we receive, if the temperature of a medical container is greater than its temperature threshold, we search the corresponding supervisor and the ALERT SERVICE he or she uses in the SQL Server database. We issue a call, from imperative code, to the sendSMS method from the ALERT SERVICE. The implementation comprises an entire application. The LINQ continuous query selects temperature notifications from medical containers that exceed temperature thresholds and calls the sendSMS method of the ALERT SERVICE of the corresponding supervisor. One insert and one delete Transact-SQL queries are used to update the SQL Server table holding dynamically discovered ALERT SERVICES. A cache is used to speed up the retrieval of temperature thresholds and container supervisors.

SoCQ. The development of this task in SoCQ contains one Serena query (Listing 4) that combines static data (temperature thresholds), method invocations (sendSMS method from SUPERVISORMOBILE) and data streams (temperatureNotification streams from supervised medical containers).

```
CREATE VIEW STREAM temperatureSupervision (mcID STRING,
    temperatureDate DATE, temperatureValue REAL,
    temperatureThreshold REAL, temperatureSent BOOLEAN)
AS SELECT mc.mcID, mc.temperatureDate, mc.temperatureValue,
    s.temperatureThreshold, mob.alertSent
STREAMING UPON insertion
FROM MedicalContainer mc, Supervise s, SupervisorMobile mob
WITH mob.alertDate := mc.temperatureDate,
    mob.alertMessage := concat("Temperature error : ",
        toString(mc.temperatureValue) )
WHERE mc.mcID = s.mcID
    AND mob.mobileID = s.mobileID
    AND s.temperatureThreshold < mc.temperatureValue
USING mc.temperatureNotification [1], mob.sendSMS;
```

Listing 4: Temperature supervision query in SoCQ

Deployment. *StreamInsight++*. This task requires the prerequisites from Task 0, as well as an instance of SQL Server.

SoCQ. Only the prerequisites from Task 0 are required.

Performance. *StreamInsight++ and SoCQ*. We evaluate all the metrics from the performance stage.

Evolution. *StreamInsight++ and SoCQ.* The user may ask to send notifications for a subgroup of the supervised medical containers. In both approaches, filters need to be added, to the imperative application, for StreamInsight++ or the Serena SQL query, for SoCQ.

Task discussion. StreamInsight++ outperforms SoCQ on the AvgLatency and MaxNoDataSources performance metrics (Table 5), which is not surprising, since the former is an ad hoc framework based on a commercial product, whereas SoCQ is a research prototype. As the service data rate increases, SoCQ outperforms our StreamInsight++ implementation when assessing MaxDataRate, due to the high number of ALERT SERVICE calls per second the query has to perform, for which SoCQ has a built-in asynchronous call mechanism. Development, deployment and evolution are easier with SoCQ, which requires no imperative code, decreased development time and a smaller number of servers and dependencies. Unlike SoCQ, StreamInsight++ does not offer an operating system independent solution. This task cannot be implemented with StreamInsight because it needs dynamic service discovery capabilities.

Table 5. Task 4 metrics

Stage	Metric	SI	SI++	SoCQ
Development	LinesOfImperativeCode	NA	175	0
	NoOfDeclarativeElements	NA	13	7
	NoOfQueries	NA	4	1
	NoOfLanguages	NA	3	1
	DevelopmentTime	NA	10	3
Deployment	NoOfServers	NA	2	1
	NoOfSystemDependencies	NA	3	1
	IsOSIndependent	NA	No	Yes
Performance	MaxNoDataSources	NA	3000	2500
	MaxDataRate	NA	275	400
	NoOfEvents	NA	13170	23812
	AvgLatency	NA	6.25	34.37
Evolution	ChangedImperativeCode	NA	1	0
	ChangedDeclarativeElements	NA	0	1

We described the SoCQ queries from Listings 1, 2b, 3 and 4, with some modifications, in the ColisTrack paper as well [8].

6 Discussion

The StreamInsight approach revealed the shortcomings encountered when developing pervasive applications with a DSMS. Such systems don't consider services as first-class citizens, nor provide dynamic service discovery. External functions

can be developed to emulate this integration in DSMSs, requiring ad hoc programming and sometimes intricate interactions with the query optimizer. With StreamInsight we were able to fully implement only Task 0 and Task 1.

StreamInsight++ was our proposed ad hoc solution for pervasive application development. The integration of different programming paradigms (imperative, declarative and network protocols) was tedious. Developing pervasive applications turned out to be a difficult and time-consuming process, which required either expert developers with more than one core area of expertise or using teams of developers. Either way, the development costs increase. Ad hoc programming led to StreamInsight++, which could be considered as a PEMS, since it handles data and services providing streams and functionalities in a pervasive environment. However, apart from the cost issues, this system carries another problem: it is specific to the pervasive environment it was designed for. A replacement of this environment automatically triggers severe changes in the implementation of the system. Moreover, although there are DSMSs which offer ways of homogeneously interacting with classical data relations and streams, in StreamInsight++ we needed a separate repository to hold static data, i.e., an instance of SQL Server.

The SoCQ PEMS solved the complex interactions between various data sources, by providing an integrated management of distributed services and a declarative definition of continuous interactions. In SoCQ we wrote declarative queries against dynamically discovered, distributed data services, the system being able to handle pervasive environments, without modifications in its implementation, as long as the services access mechanisms don't change. The price to pay was represented by the training time dedicated to the SoCQ system and the Serena SQL-like language (almost negligible for SQL developers), the description of a scenario-specific schema in Serena and the service wrappers development. Once Task 0 was accomplished, application development became straightforward. Writing SoCQ SQL-like queries was easy for someone who knew how to write SQL queries in a classical context. By comparison, the time required to study the StreamInsight platform, even if the developer had a confident level of C# and LINQ, was considerably higher. SoCQ led to concise code for Tasks 1 - 4, outperforming StreamInsight and StreamInsight++ in this respect.

The StreamInsight-based systems generally yielded better scalability and performance than SoCQ when evaluating average latency, the maximum number of data sources, or the maximum data rate. One case when SoCQ did better than the StreamInsight++ ad hoc framework, was in Task 4, when the engine had to call external services' methods at a high data rate. When assessing performance for StreamInsight and StreamInsight++, we considered the StreamInsight engine together with the adapter framework and middleware we implemented, and not the pure performance of the StreamInsight engine.

SoCQ required only one SQL-like language to write complex continuous queries over data, streams and functionalities provided by services. In the StreamInsight and StreamInsight++ implementations, an application was developed in imperative code, to execute continuous queries on the server. The only host lan-

guage allowed in the release we used (StreamInsight V1.2) is C#. SoCQ did not burden the developer with such requirements. One SoCQ server and a Java Virtual Machine were required in the SoCQ implementation and the solution could be deployed on any operating system. The StreamInsight++ solution also required more system-specific dependencies and it could only be deployed on Windows machines.

Task evolution was straightforward with SoCQ. Entities of type XD-Relation could be created to represent new service types in the pervasive environment and changes to continuous or one-shot queries had a minimal impact on the declarative code. With StreamInsight and StreamInsight++, task evolution became cumbersome, impacting imperative code. For the StreamInsight-based implementations, task evolution had an associated redeployment cost, since the code had to be recompiled.

SoCQ allows the developer to write code that appears to be more concise and somewhat elegant than the code written using the two other systems. Developers can fully implement Tasks 1 - 4 using only declarative queries. The StreamInsight and StreamInsight++ systems require imperative code as well for the same tasks, which need to be coded using an editor like Visual Studio. The imperative paradigm also adds an extra compilation step.

7 Conclusion and Future Directions

In this paper we have tackled the difficult problem of evaluating the easiness of data-centric pervasive application development. We introduced P-Bench, a benchmark that assesses easiness in the development, deployment and evolution process, and also examines performance aspects. To the best of our knowledge, this is the first study of its kind. We assessed the following approaches to building data-centric pervasive applications: (1) the StreamInsight platform, as a DSMS, (2) ad hoc programming, using StreamInsight++, an enriched version of StreamInsight and (3) SoCQ, a PEMS. We defined a set of five benchmark tasks, oriented towards commonly encountered requirements in data-centric pervasive applications. The scenario we chose can easily be changed, and the task's objectives are defined in a generic, scenario-independent manner.

We evaluated how hard it is to code a pervasive application using a set of metrics thoroughly defined. As expected, our experiments showed that pervasive applications are easier to develop, deploy and update with a PEMS. On the other hand, the DSMS- and ad hoc-based approaches exhibited superior performance for most of the tasks and metrics. However, for pervasive applications like the ones in our scenario, the PEMS implementation of the benchmark tasks achieved very good performance indicators as well. This is noteworthy, as the SoCQ PEMS is a research prototype developed in a lab, whereas StreamInsight is a giant company's product.

Future research directions include finalizing our accuracy checking framework, considering error management and resilience, data coherency, and includ-

ing additional metrics like application design effort, software modularity and collaborative development.

References

1. Abiteboul, S., Manolescu, I., Taropa, E.: A Framework for Distributed XML Data Management. In: EDBT'06, pp. 1049–1058 (2006)
2. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: The Stanford Stream Data Manager. IEEE Data Eng. Bull. 26(1), 19–26 (2003)
3. Arasu, A., Cherniack, M., Galvez, E. F., Maier, D., Maskey, A., Ryvkina, E., Stonebraker, M., Tibbetts, R.: Linear Road: A Stream Data Management Benchmark. In: VLDB'04, pp. 480–491 (2004)
4. Creus Tomàs, J., Amann, B., Travers, N., Vodislav, D.: RoSeS: A Continuous Query Processor for Large-Scale RSS Filtering and Aggregation. In: CIKM'11, pp. 2549–2552 (2011)
5. Cuevas-Vicenttín, V., Vargas-Solar, G., Collet, C.: Evaluating Hybrid Queries through Service Coordination in HYPATIA (demo). In: EDBT'12, pp. 602–605 (2012)
6. Fenton, N. E., Pfleeger, S. L.: Software Metrics: A Rigorous and Practical Approach. PWS Publishing Co., Boston (1998)
7. Gray, J.: Benchmark Handbook: For Database and Transaction Processing Systems, Morgan Kaufmann Publishers Inc., San Francisco (1992)
8. Gripay, Y., Laforest, F., Lesueur, F., Lumineau, N., Petit, J.-M., Scuturici, V.-M., Sebahi, S., Surdu, S.: ColisTrack: Testbed for a Pervasive Environment Management System (demo). In: EDBT'12, pp. 574–577 (2012)
9. Gripay, Y., Laforest, F., Petit, J.-M.: A Simple (yet Powerful) Algebra for Pervasive Environments. In: EDBT'10, pp. 359–370 (2010)
10. Kazemitabar, S. J., Demiryurek, U., Ali, M. H., Akdogan, A., Shahabi, C.: Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight. PVLDB. 3(2), 1537–1540 (2010)
11. Meijer, E.: The World According to LINQ. Commun. ACM. 54(10), 45–51 (2011)
12. Pugh, W.: Technical Perspective: A Methodology for Evaluating Computer System Performance. Commun. ACM. 51(8), 82–82 (2008)
13. Rys, M.: Scalable SQL. Commun. ACM. 54(6), 48–53 (2011)
14. Scuturici, V.-M., Surdu, S., Gripay, Y., Petit, J.-M.: UbiWare: Web-Based Dynamic Data & Service Management Platform for AmI. In: Middleware'12, pp. 11:1-11:2 (2012)
15. International Telecommunication Union: ITU Internet Reports. The Internet of Things. International Telecommunication Union (2005)
16. Weiser, M.: The Computer for the 21st Century. Scientific American. 265(3), 94–104 (1991)
17. Xue, W., Luo, Q.: Action-Oriented Query Processing for Pervasive Computing. In: CIDR'05, pp. 305–316, (2005)
18. StreamBase, `http://www.streambase.com/`
19. Microsoft StreamInsight 1.2, `http://technet.microsoft.com/en-us/library/hh849326(v=sql.10).aspx`
20. Transaction Processing Performance Council, `http://www.tpc.org`
21. Wheeler, D., Counting Source Lines of Code (SLOC), `http://www.dwheeler.com/sloc/`