End-to-end Graph Mapper

Benjamin Billet, Mickaël Jurret, Didier Parigot, Patrick Valduriez

End-to-end Graph Mapper

Benjamin Billet
Inria, Sophia-Antipolis, France
benjamin.billet@inria.fr

Mickaël Jurret
Beepeers, Sophia-Antipolis, France
mickael.jurret@beepeers.fr

Didier Parigot
Inria, Sophia-Antipolis, France
didier.parigot@inria.fr

Patrick Valduriez
Inria, Sophia-Antipolis, France
patrick.valduriez@inria.fr

ABSTRACT

The growth of linked data in web and mobile applications motivates software developers to model their business data as graphs, enabling them to leverage the capabilities of various graph databases. Going one step further, we introduce an End-to-end Graph Mapper (EGM) for modeling the whole application as (i) a set of graphs representing the business data, the in-memory data structure maintained by the application and the user interface (tree of graphical components), and (ii) a set of standardized mapping operators that maps these graphs with each other. As a benefit, the application becomes a complex live query over multiple graph databases, making the development process simpler and safer, thanks to the automation of repetitive development tasks.

This work is done in collaboration with Beepeers (`www.beepeers.com`), a startup that develops and markets social network mobile applications for small communities.

CCS CONCEPTS

• **Information systems** → Graph-based database models; • **Software and its engineering** → Model-driven software engineering;

KEYWORDS

Graph Databases; Object-Graph Mapping; Linked Data; Software Development

1 INTRODUCTION

Nowadays, an increasing number of mobile and web applications deal with linked data (e.g., social networks, online stores, recommendation systems) which leads developers to model their data as graphs. In this context, it is natural to use a Graph Database Management System (GDBMS) as an alternative to a relational DBMS, since it provides (i) dedicated data structures for storing nodes, links and key/value pairs efficiently, and (ii) query engines for browsing these structures easily [7].

In practice, web and mobile applications are typically composed of a client part and a server part that communicate with each other using web services. In a nutshell, the client part provides a graphical interface for the user while the server part manages the application logic and communicates with a standalone database for storing the linked data. As illustrated in Figure 1, each part of the application relies on dedicated data structures managed by specialized data management systems: for example, the user interface manages a tree of queryable graphical components (e.g., the document object

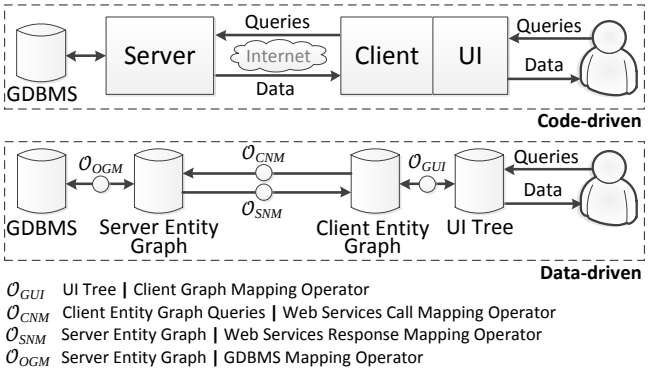


Figure 1: Representing applications as mapping operators.

model[1], in the context of web interfaces), the client application maintains a queryable cache containing offline server data or a data store specialized for specific types of applications (e.g., the knowledge base in Yarta) [9], and the server application manages queryable in-memory entities that are mapped onto actual database entities (e.g., object-relational mappers) [8].

The development of such applications is typically done by writing code or models, which tends to be tedious and error-prone given the repetitive scenarios that must be implemented by the developers. For example, two common scenarios consist in (i) filling a user interface with data retrieved from the server while maintaining a local cached version of these data in case of network failure and (ii) sending data to the server when the user interacts with the application while maintaining a stack of redoable actions in case of network failure.

As a solution, we introduce a data-driven approach, called End-to-end Graph Mapping (EGM), where (i) all the specialized data structures are materialized views of a larger dataset queried by end users, this dataset being stored by the GDBMS, and (ii) the whole application is modeled as a live query over these views. The query itself is a composition of dedicated *mapping operators* that automatically transforms data from one view to another, based on the view schemas and a set of mappings between these schemas. As a benefit, the main task of the developers consists into providing the data source schemas and the mappings between them. Thanks to the schemas, the constraints defined for each data type are always ensured at each step of the application (both client- and server-side), thus reducing the risk of inadvertent errors. Thanks to the

[1] https://www.w3.org/DOM

mappings, the repetitive development tasks are automated by the mapping operators that automatically transforms the data between the application parts.

The remainder of this paper is organized as follows. Section 2 discusses related work. Section 3 illustrates the core concepts and capabilities of our EGM approach, through two simple application use cases. Section 4 discusses the current EGM implementation. Section 5 concludes with our perspective for future work.

2 RELATED WORK

Many engineering techniques have been proposed for improving the efficiency of application development, based on declarative approaches. Such solutions are typically based on the translation of domain-specific languages [6] or models [3] into the actual application code. These approaches are designed for code production and the underlying concepts (class modeling, formal grammar, languages) are strongly tied to software development.

In contrast, instead of focusing on the code and its modeling, our approach emphasizes the data aspects and how they are queried and processed by the various data stores that compose the application (store of graphical components, caches, graph databases, domain-specific data stores, mapped in-memory data structures, etc.). From this perspective, our work can be related to multistore systems, i.e., systems that provide integrated access to heterogeneous data stores through one or more query languages [2]. Our approach indeed abstracts various specialized data sources, but introduces a set of dedicated operators used to represents the whole application as a live query over the abstracted data..

3 END-TO-END GRAPH MAPPING

Our EGM specifically targets applications composed of a client part and a server part. The server-side application includes:

- A standalone GDBMS that maintains a *full graph*, i.e., the whole set of linked data.
- A *server graph*, i.e., a graph of in-memory entities that are mapped on the actual GDBMS entities.
- Various web services for reading/updating the server graph (and, by extension, the full graph).

Similarly, the client-side application includes:

- A set of UI screens with which the user interacts.
- A *client graph*, which is a simplified version of the full graph. This client graph is maintained by the mobile application to remain usable when the network is not available (downgraded mode).

To illustrate the capabilities of our EGM, we consider a simple application use case throughout this paper. In essence, this application enables users to watch video talks and evaluate them by (i) providing comments or "likes" and (ii) by answering small sets of questions. Two scenarios are considered for covering the concepts of the EGM:

- *Scenario 1*: the user opens a talk page that displays a brief summary of the talk, the video file, the questionnaire results and the last comments.

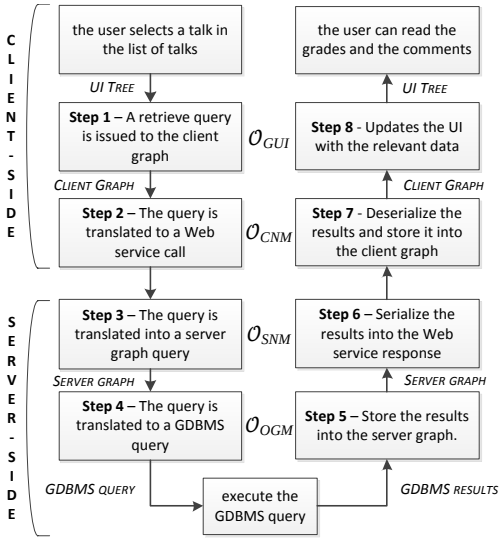


Figure 2: Typical application flow.

- *Scenario 2*: once a talk page is opened, the user can provide a comment, give a "like" to the talk or answer the questionnaire.

As an example, Figure 2 illustrates Scenario 1: when a talk page is opened, (1) a retrieve query is issued to the client graph. If the data are not available locally, (2) the query is translated into a web service call, asking the server to return the data associated to this talk. The server (3) issues a retrieve query to the server graph and (4) this query is translated into the query language of the GDBMS. The GDBMS processes the query and returns a set of results (e.g., a sub-graph). The server (5) transforms these results into in-memory entities that are stored into the server graph, (6) serializes these entities using an exchange format (e.g., JSON, XML) and sends the serialized results to the client. Finally, the client (7) deserializes these results into the client graph and (8) fills the UI screen with the relevant attributes of the Talk node (e.g., the attributes that have changed).

This kind of scenario is very common in mobile and web application development and our EGM enables developers to replace each step by a well-defined mapping operator between two graphs. These steps indeed consume and produce graph-oriented data (full graph, client graph, server graph) or tree-oriented data (user interface, JSON documents) that can be mapped with each other. Our EGM defines and standardizes the mapping operators required to write such scenarios and provides reusable implementations of these operators. Once all the mapping operators are parameterized with the proper information (typically a set of mapping definitions), the application can be represented as a sequence of graph mappings between the user interface and the full graph.

The remainder of this section will describe the mapping operators introduced by our EGM for client-side (mapping the user interface to the client graph), server-side (mapping the GDBMS content to the server graph) and between the two sides.

```
Talk:
  attributes: {title, string}, {startTime, date},
              {endTime, date}, {summary, string}
User:
  attributes: {name, string, mandatory}
Evaluation:
  attributes: {comment, string}, {like, boolean},
              {question1, boolean}, ...
  links:      {evaluated, Talk, one-to-one},
              {evaluator, User, one-to-one}
```

Figure 3: Example of full graph schema.

3.1 Full Graph Modeling

In our approach, the full graph is modeled as a directed graph where each node is defined by a node type, a *unique vertex identifier* (VID) and a set of attribute values, while each link is defined by a label. The *full graph schema* describes each node type by a name, a set of attributes and a set of outgoing links. An attribute is specified by a name, a primitive type (number, string, boolean, date, array), a value domain, a default value and some flags for expressing additional constraints regarding the attribute (e.g., "mandatory", "read-only"). A link is specified by a name, a type of destination node and a cardinality (minimum and maximum number of links).

The full graph schema associated to our application use case is illustrated by Figure 3, which defines (i) a simple set of attributes for the types Talk, User and Evaluation and (ii) the links from an Evaluation node to one Talk node (evaluated) and one User node (evaluator).

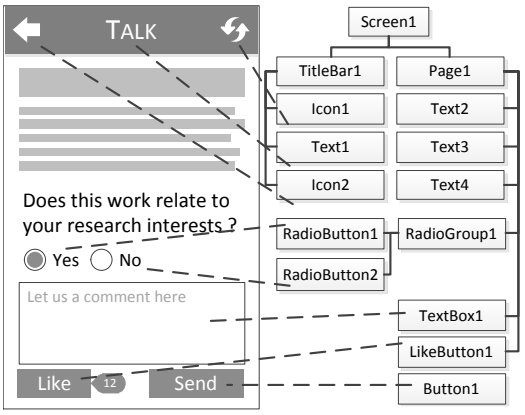
3.2 Mapping the Client Graph and the User Interface

Steps 1 and 8 of Figure 2 represents the mappings between the client state and the user interface. For example, when the user comments a talk, we want to insert automatically the comment text into the client graph when the user validate. Given that a user interface is typically modeled as a UI tree (e.g., the document object model for web pages), our EGM defines Step 1 as a mapping operator O_{GUI} from the UI tree to the client graph. Step 8 is defined as a reverse mapping operator O^{-1}_{GUI}, that applies the reverse transformation of O_{GUI} by mapping the client graph to the UI tree.

Both operators take a set of mapping definitions to operate properly. Figure 4 illustrates such definitions for O_{GUI}, based on Scenario 1. In this definition, the content of a text input (TextBox1), the state of a like button (LikeButton1) and the value of a radio group (RadioGroup1) are mapped to the "comment", "like" and "question1" attributes of the Evaluation node type.

When the user fills and validates the form by clicking on a button (Button1), the value of each mapped graphical component will be inserted in a new Evaluation node, according to the query mapped to the button.

Similarly to O_{GUI}, O^{-1}_{GUI} takes a set of additional mappings to perform the transformation from the client graph to the user interface. As shown in Figure 4, a custom expression maps the number of Evaluation nodes containing a like to the like button counter (LikeButton1.count).



(a) UI Component Tree

O_{GUI}:
```
TextBox1.text = Evaluation.comment
RadioGroup1.value = Evaluation.question1
LikeButton1.isActive = Evaluation.like
Button1.click = insert TextBox1.text, RadioGroup1.value, LikeButton1.isActive
```
O^{-1}_{GUI}:
```
LikeButton1.count = count(select in(Evaluation) where like = 1)
```

(b) Example of mapping definitions for O_{GUI} and O^{-1}_{GUI}

Figure 4: Example of user interface and client state mappings.

3.3 Mapping the Server Graph and the Full Graph

When a web service provided by the server is invoked, queries for reading or updating the server graph are issued (Step 3). These queries are automatically translated into GDBMS-specific queries for reading or updating the full graph (Step 4). The GDBMS processes the queries and returns a set of results that are converted into server graph entities (Step 5). These entities are serialized into an exchange format (Step 6) and sent back to the client for updating the client graph.

In applications based on relational databases, the mapping between the server application and the standalone database can be managed automatically by an object-relational mapper, which maps the database types to the data structures of the language the server application is written with (e.g., Java classes). Similarly, our EGM includes an object-graph mapper [4] for mapping the types of the full graph to the types of the server graph.

Step 4 is performed by the mapping operator O_{OGM} parameterized with a set of definitions that maps each type of the server graph to the corresponding type of the full graph, enabling the server to build the relevant GDBMS requests. The reverse mapping operator O^{-1}_{OGM} uses the same definitions for mapping the query results types to the server graph types.

Figure 5 illustrates a O_{OGM} definition for Scenario 1, where three Java classes, called Talk, Evaluation and User, are mapped to the Talk, Evaluation and User types of the full graph schema.

```
class Talk {
    private String talkTitle;
    ...
    private List<Evaluation> evaluations;
    private List<User> evaluators;
}
class Evaluation {
    private Boolean like;
    private String comment;
    private Boolean question1;
    private Talk talk;
    private User user;
}
class User {
    private String name;
    private List<Talk> evaluatedTalks;
}
```

(a) Java classes

O_{OGM} :

```
  Talk.class:
    graphType:  Talk
    attributes: {talkTitle, title}, ...,
    links:      {evaluations, in(evaluated)}
                {evaluators, in(evaluated).evaluator}
    autofetch:  in(evaluated)
    autodelete: in(evaluated)
  Evaluation.class:
    graphType:  Evaluation
    attributes: {liked, like}, {comment, comment}, {question1, question1}
    links:      {talk, out(evaluated)}
    autofetch:  out(evaluated)
  User.class:
    graphType:  User
    attributes: {name, name}
    links:      {evaluatedTalks, out(evaluator).evaluated
    autodelete: in(evaluator)
```

(b) Example of mapping definitions for O_{OGM}

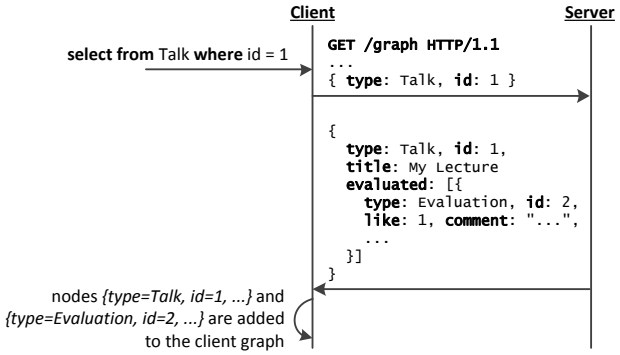
Figure 5: Example of server graph and full graph mappings.

3.4 Client-server mappings

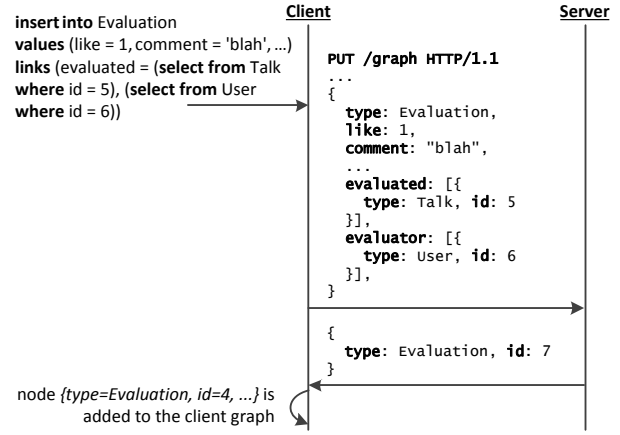
Steps 2, 3, 6 and 7 are related to the communication between the client part and the server part. Several mapping operators are involved in this communication: O_{CNM} manages the invocation of web services and O_{CNM}^{-1} deserializes the query results returned by the server as a response to the web service call, while O_{SNM} translates web services calls into server graph queries and O_{SNM}^{-1} serializes the results of these queries into a tree-oriented exchange format, such as JSON or XML.

The typical behavior of O_{CNM} consists into translating client graph queries into web services calls carrying a *request graph* that must be completed or merged with the content of the full graph, depending on the type of the query:

- **Retrieve Queries** (e.g., `select from Talk`)
  A GET request is issued to the server and the request graph represents a pattern to match with the server graph.
- **Create Queries** (e.g., `insert into User values (...)`)
  A PUT request is issued to the server and the request graph must be added to the server graph.
- **Update Queries** (e.g., `update Evaluation set like=0`)
  A POST request is issued to the server and the request graph must be merged with the server graph.
- **Delete Queries** (e.g., `delete from Evaluation`)
  A DELETE request is issued to the server and the request graph represents a sub-graph that must be deleted from the server graph.



(a) Scenario 1



(b) Scenario 2

Figure 6: Example of request graphs exchanged between client and server.

When the server process the web service call, O_{SNM} translates the request graph into a query for the server graph, by mapping each type, attribute and link of the request graph to the corresponding type, attribute and link of the server graph. The results of the translated query are managed by O_{SNM}^{-1}, which serializes the set of result nodes into a tree structure that is sent back to the client. After receiving the serialized results, O_{CNM}^{-1} deserializes them and updates the client graph accordingly.

Figure 6 illustrates the behavior of these mapping operators for Scenarios 1 and 2, by displaying the HTTP requests exchanged between the client end the server.

In Scenario 1, the user selects a talk in a talk list. This action triggers a query for retrieving the Talk node with the corresponding VID in the client graph. If the node is not already in the client graph, O_{CNM} translates the query into a GET request that embeds a request graph representing the pattern that must be matched by the server. The server processes the request and sends back the data associated to the requested Talk node, i.e., the node attributes and all the linked Evaluation nodes, as specified by the fetch plan

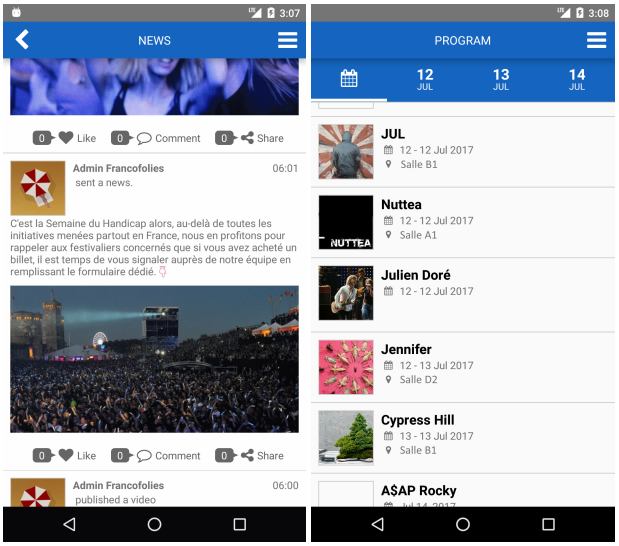


Figure 7: Mobile Application Screenshots

defined in Figure 5. Finally, the client merges the results with the client graph, potentially triggering changes in the user interface.

In Scenario 2, the user evaluates the talk. This action triggers a query for inserting a new Evaluation node filled using the data input by the user through the user interface. A new Evaluation node is inserted into the client graph, without a VID. As a reaction, O_{CNM} translates the query into a PUT request that embeds a request graph representing the node and the links to create. When the server processes the request graph, it creates the new Evaluation node and links it to the existing User and Talk nodes. As a response, the server sends back the VID associated to the new Evaluation node in the full graph and the client updates the client graph accordingly. At this point, the client graph and the full graph are correctly synchronized.

4 CURRENT IMPLEMENTATION STATUS

Our work takes place in the context of a joint project with Beepeers (`www.beepeers.com`), a company that develops and markets social network mobile applications for small communities, i.e., communities that share common activities and interests (e.g., associations, companies) or attend to a same event (e.g., conference, concert). The development of our EGM prototype is based on the common scenarios identified in a set of social network applications (approximately 100 concepts in the full graph) actually developed by Beepeers.

The client side is implemented using Javascript technologies. The user interface is based on *React* (http://facebook.github.io/react) and *React-Native* (http://facebook.github.io/react-native), two framework for managing user interfaces components and their states, based on the document object model (web applications) or on native mobile UI frameworks (mobile applications). The client graph is implemented using *Redux* (http://redux.js.org), a framework for managing a global application state and linking it to the user interface: when an action updates a part of the global state, the graphical components that are connected to this part are automatically updated.

The server side is implemented using Java technologies. The server graph is based on *TinkerPop* (http://tinkerpop.apache.org), a generic framework for dealing with graph data. TinkerPop provides Object-Graph Mapping capabilities and a common interface to access standalone graph databases. Finally, the full graph is stored into an *OrientDB* (http://orientdb.com) database, a multi-model (document and graph) database that supports inheritance.

Our prototype is still work in progress, where the mapping operators are currently integrated into a small customized code base. From our experiments, the graph query languages are sometimes not enough to express any type of computation and the developers may want to customize the operators by injecting specific code.

5 CONCLUSION

In this paper we proposed the concept of End-to-End Graph Mapper: a set of mapping operators for representing web and mobile applications as (i) a multistore system composed of dedicated graph databases and (ii) live queries over this multistore system, in such a way that the graph data are mapped from one database to another. This work is still in progress and we plan to extend it in several direction.

From a technical perspective, our operators can be improved to deal with additional non-functional use cases, such as user authentication and access control to the full graph data: new concepts must be added to our schema and mapping definition languages in order to enforce security constraints automatically at each step of the application. Similarly, we want to investigate how the mapping functions could support transactions and how our EGM could support replayable updates in case of sparse network connectivity. Second, we want to investigate alternative implementations for our EGM. Currently, it is implemented as a set of reusable operators that consumes mapping definitions at runtime. If an operator does not suit a use case, the developers must write a new operator or deal with the current operator interfaces to inject custom code. Another solution consists into generating the application source code, by building automatically an application-specific implementation of operators based on the mapping definitions. In this case, the application source code would be fully customizable by the developers.

From a functional perspective, we plan to extend this work to integrate other types of applications (e.g., desktop applications), architectures (e.g., decentralized architectures instead of client-server) [5] and communication paradigm (e.g., data streaming through services) [1]. Given that our approach is extensible, other types of architecture could be implemented by reusing some of the mapping operators presented in this paper or by introducing new dedicated operators. In addition, dealing with various types of applications and architectures would enable us to investigate more use cases in order to improve the query and mapping definition languages introduced in this paper.

REFERENCES

[1] Benjamin Billet, Valerie Issarny, and Géraldine Texier. 2017. Composing Continuous Services in a CoAP-based IoT. In *Proc. of 2017 IEEE International Conference on AI & Mobile Services.*

[2] Carlyna Bondiombouy and Patrick Valduriez. 2016. *Query Processing in Multistore Systems: an overview.* Technical Report. INRIA Sophia Antipolis - Méditerranée.

[3] Alberto Rodrigues da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43 (2015).
[4] Felix Dietze, Johannes Karoff, André Calero Valdez, Martina Ziefle, Christoph Greven, and Ulrik Schroeder. 2016. *An Open-Source Object-Graph-Mapping Framework for Neo4j and Scala: Renesca.* Springer.
[5] Fady Draidi, Esther Pacitti, Didier Parigot, and Guillaume Verger. 2011. P2Prec: A Social-based P2P Recommendation System. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management.*
[6] Dean Kramer, Tony Clarl, and Samia Oussena. 2010. MobDSL: A Domain Specific Language for multiple mobile platform deployment. In *Proc. of 2010 IEEE International Conference on Networked Embedded Systems for Enterprise Applications.*
[7] Montiago Labute and Matthew Dombroski. 2014. *Review of Graph Databases for Big Data Dynamic Entity Scoring.* Technical Report. Lawrence Livermore National Laboratory.
[8] Craig Russell. 2008. Bridging the Object-Relational Divide. *Queue* 6, 3 (2008).
[9] Alessandra Toninelli, Animesh Pathak, and Valérie Issarny. 2011. Yarta: A Middleware for Managing Mobile Social Ecosystems. In *Proc of the 2011 International Conference on Advances in Grid and Pervasive Computing.* Springer.