Process-Driven Biometric Identification by means of Autonomic Grid Components

Int. J. of Autonomous and Adaptive Communications Systems, Vol. x, No. x, xxxx
1

Process-Driven Biometric Identification by means of Autonomic Grid Components

Thomas Weigold*

IBM Zurich Research Lab., Zurich, Switzerland
E-mail: twe@zurich.ibm.com
*Corresponding author

Marco Aldinucci

Computer Science Department, University of Torino, Italy
E-mail: aldinuc@di.unito.it

Marco Danelutto

Computer Science Department, University of Pisa, Italy
E-mail: marcod@di.unipi.it

Vladimir Getov

School of Electronics and Computer Science,
University of Westminster, London, U.K.
E-mail: V.S.Getov@westminster.ac.uk

Abstract: Today's business applications are increasingly process driven, meaning that the main application logic is executed by a dedicate process engine. In addition, component-oriented software development has been attracting attention for building complex distributed applications. In this paper we present the experiences gained from building a process-driven biometric identification application that makes use of Grid infrastructures via the Grid Component Model (GCM). GCM, besides guaranteeing access to Grid resources, supports autonomic management of notable parallel composite components. This feature is exploited within our biometric identification application to ensure real time identification of fingerprints. Therefore, we briefly introduce the GCM framework and the process engine used, and we describe the implementation of the application by means of autonomic GCM components. Finally, we summarize the results, experiences, and lessons learned focusing on the integration of autonomic GCM components and the process-driven approach.

Keywords: autonomic computing, components, parallel applications, distributed applications, process-driven applications

Reference to this paper should be made as follows: T. Weigold, M. Aldinucci, M. Danelutto, and V. Getov, 'Development Methodology for Autonomic Process-Driven Business Applications', *Int. J. of*

Autonomous and Adaptive Communications Systems, Vol. x, No. x, pp.xxx–xxx.

Biographical notes: Thomas Weigold received his MSc in Computer Science 2002 from the University of Westminster, London. He is a software engineer in the Department of Computer Science, IBM Zurich Research Laboratory, Switzerland. His research interests include workflow and business process management, distributed computing, and secure systems.

Marco Aldinucci received his PhD in Computer Science 2003 from the University of Pisa. He his an assistant professor in the Department of Computer Science, University of Torino, Italy. His research interests include adaptive and autonomic computing, languages and tools for parallel computing, multi- and many-core processing.

Marco Danelutto got his PhD in Computer Science in 1990 and is currently an associate professor at the Dept. of Computer Science, Univ. of Pisa. His research interests include structured parallel programming models, algorithmic skeletons and autonomic computing for parallel and distributed architectures (multi/many-core, clusters and grids).

Vladimir Getov is a professor of distributed and high-performance computing at the University of Westminster, London. His research interests include parallel architectures and performance, autonomous distributed computing, and high-performance programming environments. He received a PhD and DSc in Computer Science from the Bulgarian Academy of Sciences.

1 Introduction

Today's businesses are increasingly process driven. Ideally, all actions within an enterprise are explicitly defined as processes with the goal to improve control, flexibility, and effectiveness of delivering customer value. Additionally, business processes are oftentimes supported or even fully implemented by software applications (zur Muehlen 2004). In many cases, the business processes are turned into software such that they are hidden in the application's source code. However, there is a trend towards separating the main business logic from the functional code such that the resulting applications become more transparent and more flexible. The approach is to embed a so-called process engine into the application, which then executes process definitions representing the main control logic of the application. Functional code is then triggered from the process engine in accordance with the process definition. Such applications are called process-driven or workflow-driven applications. The main advantages of this approach are the fact that the application logic can be modified without re-compiling the application, even at runtime, the business logic is more evident, and monitoring features of the process engine can be explored.

Besides the trend towards process-driven applications, enterprises seek ways to benefit from resources available from computing Grids/Clouds, in particular in all those cases were parallel computing is required to guarantee fair performances. Within the plethora of programming environments targeting Grids, GCM (the Grid Component Model developed within CoreGRID project (2007) and whose reference implementation has been provided by GridCOMP project (2008)) supports Grid programmers in designing parallel/distributed grid applications. In particular, GCM provides pre-defined autonomic composite components modelling standard parallel/distributed computation patterns.

In this work, we discuss a process-driven application, which makes use of GCM autonomic components to solve the problem of large-scale biometric identification, which has been developed as part of the activities of the GridCOMP project (2008). In particular, we discuss how process-driven application development exploits the autonomic features provided by the underlying Grid software as well as the results, experiences, and lessons learned during application development focusing on the integration of autonomic GCM components and the process-driven approach.

The paper is organized as follows: Sec. 2 discusses related work with respect to the process-driven approach and autonomic Grid components. Sec. 3 introduces GCM and Behavioural Skeletons (BS). Sec. 4 introduces the process engine used to implement the biometric identification application presented in Sec. 5. Eventually, Sec. 6 discusses the overall results achieved and Sec. 7 drafts the conclusions of the paper.

2 Related Work

With the recent hype around business process management (BPM) (Smith & Fingar 2006) interest in building process-driven applications with the goal to arrive at more flexible IT systems that can keep up with a very agile business environment is increasing. However, implementing the process-driven approach in diverse application domains requires generic and easily embeddable process engines. While many of the contemporary process frameworks are rather domain specific and monolithic, a few developments into providing such generic engines can be observed (Bukovics 2007, Faura & Baeyens 2007, Weigold et al. 2007). Nevertheless, the majority of the work focuses on mapping higher-level process modelling languages such as BPEL, XPDL, or BPMN (Ryan 2009) to executable code interpreted by the process engine (Freeman & Pryce 2006). Furthermore, most of the application scenarios are rooted in the traditional workflow domain, for example, document and human task management. Therefore, in this paper we apply a generic process-engine to the problem of distributed biometric identification with the goal to evaluate how the process-driven approach integrates with advanced distributed computing frameworks supporting autonomic features, such as GCM.

The idea of autonomic management of parallel/distributed/grid applications is present in several programming frameworks, although in different flavours. ASSIST (Vanneschi 2002, Aldinucci & Danelutto 2006), AutoMate (Parashar et al. 2006), K-Components (Dowling 2004), SAFRAN (David & Ledoux 2006) and finally GCM (CoreGRID project 2007) all include autonomic management features. The latter two are derived from a common ancestor, i.e. the Fractal hierarchical component

model (Bruneton et al. 2003). All the named frameworks, except SAFRAN, are targeted to distributed applications on grids, and all except ASSIST are component based. Though these programming frameworks considerably ease the development of an autonomic application for the grid (to various degrees), all of them but GCM fully rely on the application programmer's expertise for the set-up of the management code, which can be quite difficult to write since it may involve the management of black-box components, and, notably, is tailored to the particular component or to a particular component assembly. As a result, the introduction of dynamic adaptivity and self-management might enable the management of grid heterogeneity, dynamism, and uncertainty aspects but, at the same time, decreases the component reuse potential since it further specialises components with application specific management code.

While component models provide a suitable way to *spatially* compose the parts of a applications, whereas workflow models have been mainly developed to support composition of independent programs (usually loosely coupled and named tasks) by specifying temporal dependencies among them (and defining a *temporal* composition, in fact), to support efficient scheduling onto available resources, e.g. sites, processors, memories (Fox & Gannon 2006). Recently, various attempts to merge components and workflows models into a spatio-temporal model have been undertaken, e.g. in the *Spatio-Temporal Component Model* (Bouziane et al. 2008)).

As we shall see, GCM mainly differentiate from other frameworks because it provides programmers with skeletons (i.e. high-level programming patterns) that substantially ease non-functional management of applications. It is worth noticing that the introduction the skeleton concept could equally have built upon K-Components or the AutoMate framework as all provide distributed system based component frameworks with autonomic capability.

3 The GCM framework

The *Grid Component Model* (GCM) is a component model explicitly designed within CoreGRID project (2007) to support component-based autonomic applications in distributed contexts. The main features of this component model can be summarised as follows:

- **Hierarchical:** GCM components can be composed in a hierarchical way in *composite* components. Composite components are first class components and they are not distinguishable from non-composite components at the user level. Hierarchical composition greatly improves the expressive power of the component model and is inherited by GCM from the Fractal component model (Bruneton et al. 2003).
- **Structured:** In addition to standard intra-component interaction mechanisms (use/provide ports (Armstrong et al. 1999)) GCM allows components to interact through *collective* ports modelling common structured parallel computation communication patterns. These patterns include broadcast, multicast, scatter and gather communications operating on collections of components. Also, GCM provides *data* and *stream* ports, modelling access to shared data encapsulated into components and data flow

streams. All these additional port types, not present in other well known component models, increase the possibilities offered to the component system user for developing efficient parallel component applications.

- **Autonomic:** GCM specifically supports implementing autonomic components in two distinct ways: by supporting the implementation of user defined component controllers and by providing behavioural skeletons. Component controllers can be programmed in the component membrane (the membrane concept, as the place where component control activities take place, is inherited from Fractal (Bruneton et al. 2003)) and controllers can be components themselves. This provides a substantial support to the development of reusable autonomic controllers. Behavioural skeletons, thoroughly discussed in Sec. 3.1, are composite GCM components modelling notable parallel/distributed computation patterns and supporting autonomic managers, i.e. components taking care of non functional concerns affecting parallel computation.

Due to the presence of controllers and autonomic managers, GCM components implement two distinct kinds of interfaces: functional and non-functional ones. The functional interfaces host those ports concerned with the implementation of the functional features of the component. The non-functional interfaces host the ports related to controllers and autonomic managers. These ports are the ones actually supporting the component management activity in the implementation of the non-functional features, i.e. those features contributing to the efficiency of the component in obtaining the expected (functional) results but not directly involved in result computation.

GCM has been designed within the Institute on Programming model of the CoreGRID NoE (2009) and a reference implementation of the component model has been developed within the GridCOMP project (2008). Within the same GridCOMP project, a Grid Integrated Development Environment (GIDE) has been developed to support development and maintenance of GCM programs.

3.1 Behavioural Skeletons

Behavioural skeletons (BS) represent a specialisation of the algorithmic skeleton concept for component management (Cole 2004). Algorithmic skeletons have been traditionally used as a vehicle to provide efficient implementation templates of parallel paradigms. BS, as algorithmic skeletons, represent patterns of parallel computations (which are expressed in GCM as graphs of components), but in addition they exploit the inherent skeleton semantics to design sound self-management schemes of parallel components.

BS are composed of an algorithmic skeleton together with an autonomic manager (see Fig. 1). They provide the programmer with a component that can be turned into a running application by providing the code parameters needed to instantiate the algorithmic skeleton parameters (e.g. the code of the different stages in a pipeline or the code of the worker in a task farm) *plus* some kind of Service Level Agreement (SLA, e.g. the expected parallelism degree or the expected throughput of the application). The code parameters are used to build the actual code run on the target parallel/distributed architecture, while the SLA is used by

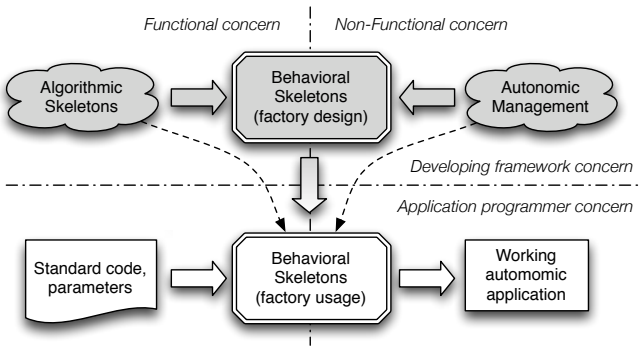

Figure 1 Behavioural skeleton rationale

the autonomic manager that will take care of ensuring this SLA (best effort) while the application is being computed.

The choice of the skeleton to be used as well as the code parameters provided to instantiate the BS are functional concerns: they only depend on what has to be computed (i.e. on the application at hand) and on the *qualitative* parallelism exploitation pattern the programmer wants to exploit. The autonomic management itself is a non-functional concern. The self-management and self-tuning activities taking place in the manager to ensure user supplied SLA both depend on the application structure (the one defined by the algorithmic skeleton) and on the target architecture at hand. The implementation of both the algorithmic skeleton and the autonomic manager is in the charge of the "system" programmer, i.e. the one providing the behavioural skeleton framework to the application user.

In the programming model provided by BS, the application programmers are in charge of picking up a BS (or a composition of BS) among those available and of providing the corresponding parameters and SLA. The system, and in particular the autonomic managers of the BS instantiated by the application programmer, are in charge of performing all those activities needed to ensure the user supplied SLA. These activities, in turn, may include varying some implementation parameters (e.g. the parallelism degree, the kind of communication protocol used among different parallel entities or scheduling/mapping of the parallel activities to the target processing elements) as well as changing the BS (composition) chosen by the application programmer (e.g. using "under the hoods" an equivalent, but more efficient (with respect to the target architecture and user supplied SLA) behavioural skeleton (composition)).

Autonomic management of non-functional concerns is based on the concurrent execution (with respect to the application "business logic") of a basic control loop such as that shown in Fig. 2. In the *monitor* phase, the application behaviour is observed, then in the *analyse* and *plan* phases the observed behaviour is examined to discover possible malfunctioning and corrective actions are planned. The corrective actions are usually taken from a library of known actions and the chosen action is determined by the result of the analysis phase. Finally, the actions planned are applied to the application during the *execute* phase (Kephart & Chess 2003, Danelutto 2005, Aldinucci & Danelutto 2006, Aldinucci et al. 2007, 2009*c*,*b*). Currently, two kind of BS are implemented in GCM: a *task farm* BS and a *data*

parallel BS (see Fig. 3). The former models embarrassingly parallel computations processing independent items x_i of an input stream to obtain items $f(x_i)$ of the corresponding output stream. The latter models data parallel computations by computing for each item of the input stream x_i an item $f(x_i, D)$ of the corresponding output stream, where D represents a read only data structure and the result of $f(x_i, D)$ can be computed as a map of some function $f'(x_i)$ on all the items of D followed by a reduce of the different $f'(x_i, D_j)$ with an associative and commutative operator g.

Both BS implement an AM taking care of the performance of the parallel computation at hand. In particular, the AM may ensure contracts stating the expected service time (or throughput, i.e. the time between the delivery of two consecutive items on the output stream) of the BS (both task farm and data parallel BS) or the expected partition size of data structure D (data parallel BS only). Currently, the contracts must be supplied to the BS AMs through the BS non functional ports as a(n ASCII string hosting a) set of JBoss rules defined in terms of the operations provided by the ABC controller bean. In fact, the AM control loop is implemented by running an instance of the JBoss business rule engine at regular intervals of time. At each time interval, all the *pre-condition-action* rules supplied to the AM are evaluated and those that turn out to be fireable (e.g. whose with the pre-condition holding true) are executed ordered by priority (or *salience* according to JBoss jargon). The pre-conditions are evaluated using values provided by the monitoring system implemented in the ABC controller beans, actually. The period used to run the JBoss engine is determined in such a way it is neither too fast (reacting when it was not the case to react to small changes in the system, thus increasing overhead to the autonomic management) nor too slow (poorly reacting to actual changes in the system, thus decreasing efficiency of autonomic management).

The AMs taking care of performance in behavioural skeletons manage the contracts varying the parallelism degree of the BS, i.e. the number of worker instances actually used to implement the BS. The variation of the number of worker instances happens adding/removing a fixed amount of workers. This fixed amount is a BS user configurable constant (Δ_w). Rules supplied to the AM in the BS also consist in specific rules avoiding to perform (probably) useless adaptations (e.g. avoiding to adapt BS parallelism degree immediately after another adaptation took place) as well as rules default actions basically only taking care of updating monitored values when no other, more significant actions turn out to be fireable.

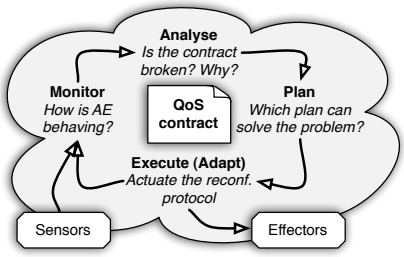

Figure 2 The classical control loop implemented within Autonomic Managers in GCM Behavioural Skeletons

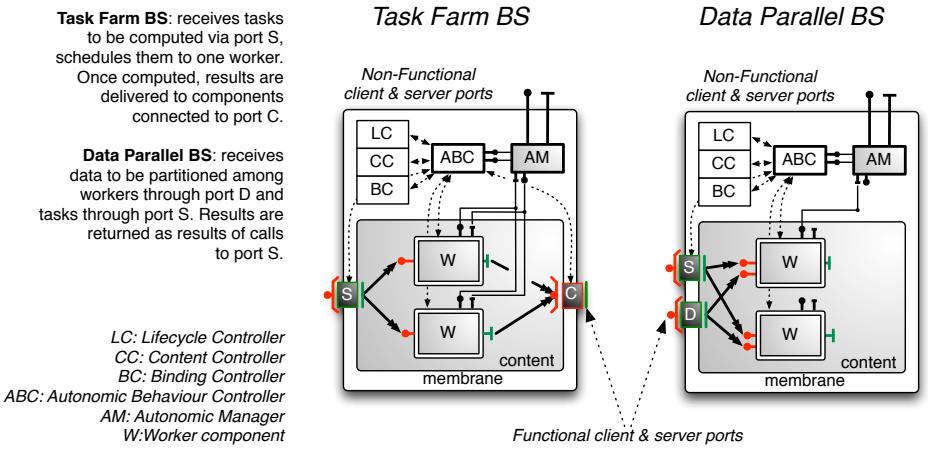

Figure 3 Behavioural skeletons currently implemented in GCM

Recently, the behavioural skeleton framework has been extended in such a way:

- The BS set has been extended, introducing a *pipeline* BS, modelling computations organized in stages.
- BS may be nested to model more and more complex applications. This implies that managers belong to a manager hierarchy reflecting the BS nesting used to model the application.
- A procedure has been developed to derive local contracts for all the BS used to implement a given application from the initial, application directed, user supplied performance contract. The top level AM manages to split the user supplied contract in subcontracts that are then passed to nested BSs. The procedure is recursively applied in such a way eventually each BS in the BS tree has is own subcontract. The subcontracts are derived in such a way if all of them are ensured, the unique, application performance contract supplied by the user is also ensured.
- Policies and contracts have been experimented that allow an AM to inform its ancestor AM and to enter a *passive* state in all those cases where no local action may be taken to enforce the local contract. The upper level manager is expected to eventually provide the "passive mode" AM with a new contract, such that this AM can start again is usual (*active* mode) control loop.

This improved BS framework has been proven to be able to suitably handle complex autonomic management policies, such as those where manager actions on BS_i are *de facto* triggered by another autonomic manager BS_j (Aldinucci et al. 2009*b*). As an example, in this extended framework, an AM of a farm being the $i-th$ stage in a pipeline could report a contract violation to the pipeline AM, not being able to sustain its throughput due to a poor input task inter-arrival rate. Therefore, the pipeline AM may send a new contract to stage i aimed at increasing its output rate, such that eventually the farm can succeed fulfilling its contract. Despite the fact

these new features are not necessary to support the case discussed in this paper, it is worth pointing out that applications more complex—in terms of the structure of parallelism exploited—of the one used in this paper may be targeted.

4 The ePVM Process Engine

The embeddable Process Virtual Machine (ePVM) is a research prototype process engine basically built upon two core concepts. Firstly, a process model that is rooted in the theoretical framework of communicating extended finite state machines (CEFSM). Secondly, whereas many efforts have been made to create the ultimate process language, ePVM provides in contrast a low-level run-time environment based on a JavaScript interpreter where higher-level domain specific process languages can be mapped to.

The idea of ePVM can be considered to follow a bottom-up or micro-kernel type of approach for building process-driven applications, Business Process Management Systems (BPM), or workflow systems. This means that ePVM is a basic framework for building such systems rather than a complete off-the-shelf application that can run stand-alone. It consists of a library including a lightweight, generic, and easily programmable process execution engine. Lightweight hereby means that the engine is small in size and imposes minimum requirements on its environment, namely the host application it is embedded in. ePVM has its own process model resembling networks of communicating state machines running in parallel, which makes it an inherently asynchronous, event-driven run-time system. Every state machine is implemented by one JavaScript function, has an associated thread executing it, has a state object which is passed every time the function is invoked, and can communicate with other processes as well as entities external to the process engine via some messaging system. An arbitrary number of external entities, so-called host processes, can be attached to the engine to become visible for ePVM processes. The ePVM programming model based on the theory of CEFSM combines the simplicity of JavaScript with an easy and powerful way of defining complex concurrent business processes. More details can be found in (Weigold et al. 2007).

5 Process-Driven Distributed Biometric Identification

In recent years biometric methods for verification and identification of people have become very popular. Applications span from governmental projects like border control or criminal identification to civil purposes such as e-commerce, network access, or transport. Frequently, biometric verification is used to authenticate people meaning that a 1:1 match operation of a claimed identity to the one stored in a reference system is carried out. In an identification system, however, the complexity is much higher. Here, a person's identity is to be determined solely on biometric information, which requires matching the live scan of his biometrics against all enrolled (known) identities. Such a 1:N match operation can be quite time-consuming making it unsuitable for real-time applications. In order to tackle this challenge, a distributed biometric identification system (BIS), which can work on a large user population of up to millions of individuals, has been developed. It

is based on fingerprint biometrics and allows real-time identification within a few seconds period by taking advantage of the Grid, in particular via GCM components.

5.1 Application Architecture

The BIS can be considered a process-driven application, as it is centrally driven by the ePVM process engine. Fig. 4 outlines its high-level architectural design.

A number of ePVM process definitions describing the main control flow for operations such as starting up the system or identifying a person are loaded into the process engine. These processes co-operate with external entities such as the GUI, the database (DB) of known identities, and the distributed GCM component system via a number of host processes to implement the overall functionality of the BIS.

5.2 Process-Engine/GCM Interfacing

The actual distributed fingerprint matching functionality is implemented via a set of GCM components deployed within a Grid/Cloud infrastructure as indicated in Fig. 4. Processes running within the process engine must be able to create, deploy, configure and interact with these components. For this purpose, a dedicated host process named *GCM adapter* (c.f. Fig. 4) has been developed, which receives messages from ePVM process instances, turns these messages into method invocations on GCM framework methods or GCM components, and generates appropriate reply messages returned to ePVM. The GCM adapter represents the main interface between ePVM and GCM. As ePVM process definitions are

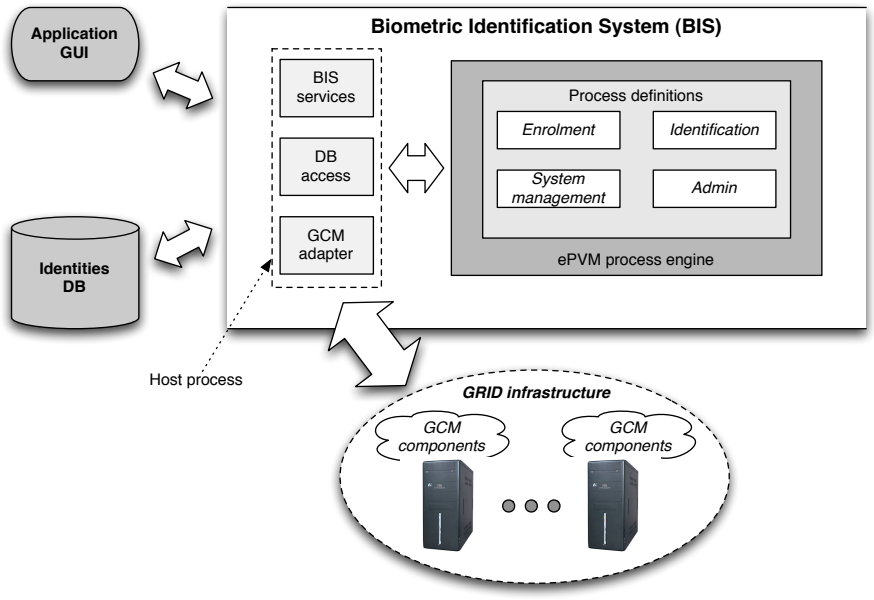

Figure 4 BIS high-level architecture

implemented in JavaScript and the GCM framework is available as a Java library, the GCM adapter essentially converts between JavaScript messages and Java method invocations.

An alternative option would have been to export the GCM components as Web services, as supported by the GCM implementation, and invoke them from within the GCM adapter. However, this would have increased the number of required type conversions going from Java Script over SOAP to Java and vice versa. Also, the GCM framework only supports exporting GCM components as Web services. Other framework services, for example, functionality for deployment and component creation, cannot be turned into Web services automatically. Finally, the ePVM process engine does not necessarily require working on Web Services level like, for instance, process engines based on the Business Process Execution Language (BPEL). Consequently, we decided not to use Web services as interfaces between the process engine and GCM.

The functionality provided by the GCM adapter includes:

- Activate a given GCM deployment descriptor to start the nodes available in the Grid.
- Modify architecture description language (ADL) files describing the GCM components used.
- Create GCM components within the Grid.
- Invoke methods on GCM components, for example, to configure the quality of service (QoS) contract, distribute the DB of known identities, or submit the biometrics of a person for identification.

The GCM adapter is triggered by ePVM process instances to implement the overall application logic. As an example, the activity flow chart shown in Fig. 5 illustrates the control logic implemented within an ePVM processes as it is executed during BIS initialization. For each of the activities a message is being sent to a host adapter which implements the functionality. Some of the activities execute in parallel, for instance, activity 1.1 to 1.3, some are sequential.

5.3 Using Autonomic GCM Components

The problem of biometric identification can be considered a search problem where the compare function is a biometric matching algorithm, here fingerprint matching. To distribute the problem within a Grid infrastructure, the DB of known identities needs to be distributed such that each computing node in the Grid receives a partition of the overall DB and can match a given identity against this partition. The time spent in matching the given identity against the local portion of the database is clearly proportional to the size of this local DB portion. Therefore, considering that the distribution of the DB among the grid nodes is performed once and for all, and considering negligible the time spent to broadcast the fingerprint that has to be matched with those in the distributed database, the ability to perform fingerprint matching in real time roughly depends on the ability to distribute local portions of the database small enough to allow real time matching of the broadcasted fingerprint. More precisely, the time spent in matching a single fingerprint against

the local database also depends on the computing power and on the load of the machine used to perform the matching. The machine power and the local database sizes are somehow static properties. The load of the machine is instead a dynamic property. Thus, in order to keep the matching time perceived by the application user within a given range (i.e. satisfying a given service level agreement (SLA) or *performance contract*), our BIS application should i) properly dimension the number of distributed resources used to host database portions and ii) dynamically adapt to the varying load of the grid resources involved in such a way a user supplied performance contract (such as *match fingerprint in less than 30 seconds*) is ensured. Both features are supported within the GCM Behavioural skeletons presented in Sec. 3.1: if the user instantiates a Behavioural skeleton to implement the BIS search process, and if he/she provides a contract stating the expected latency of the fingerprint matching process, the AM of the behavioural skeleton will start with a predefined number of workers (i.e. a predefined parallelism degree) and then adapt this number to achieve the matching latency adding (removing) workers from the BS composite component. In case of overload of some of the resources used in the matching, the AM of the behavioural skeleton will also manage to increase the number of resources recruited to the parallel matching, in such a way the contract can be ensured again. In this case, the recruitment of a new processing resource induces a physical redistribution of the database among the resources. This redistribution is completely implemented/managed by the behavioural skeleton AM.

In order to implement our BIS application, we used a data parallel (DP) behavioural skeleton. Referring to Fig. 3 (right), the DP skeleton is a composite component which includes an autonomic behaviour controller (ABC) and an autonomic manager (AM). The AM periodically evaluates certain monitored properties of the skeleton to ensure that a given QoS contract is satisfied. If this is not the case, it triggers appropriate reconfiguration operations provided by the ABC. To apply the DP skeleton for our application scenario, it must be parameterised with a worker component and a QoS contract. The worker

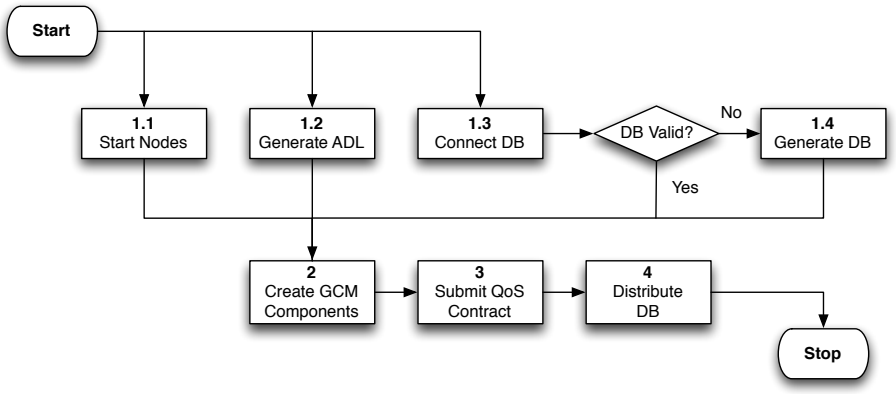

Figure 5 BIS initialization process flow

component, here named *IDMatcher*, implements the actual fingerprint matching functionality and the skeleton allocates one instance of this worker component per node. The QoS contract consists of a set of rules interpreted by the JBoss Drools rule engine.

For our BIS prototype we chose to implement a QoS contract requiring to keep the partition size of the workers constant, independently of the size of the database presented to the BS through port D. The contract is provided before starting the computation through the non functional server port attached to the BS AM. The AM, in this case, adds or removes workers from the BS in case the partition size exceeds or is less than the value supplied by the user within the contract provided through the non functional BS ports.

Before identification requests can be processed, the identity DB is distributed across the worker components using port D. As a consequence, the DB is partitioned on the inner W components. The identity DB holds information such as name, address, and fingerprints of all enrolled (known) people.

Once the skeleton has been initialized, identification requests can be submitted via the second port provided by the BS, port S, the so-called broadcast port. Fingerprints of a person to be identified are broadcasted via this port to all worker components and each worker matches them against its partition of the DB. Results are returned synchronously via method return values.

If the AM triggers reconfiguration via the ABC, for example, to increase the number of worker components, the AM collects all DB partitions from the workers, modifies the number of workers, and finally redistributes the DB to the workers. This way the DB is redistributed during each reconfiguration operation.

The submission of the contract through non functional interfaces, the DB through BS port D, and the fingerprints to be matched through port S are all interactions with the GCM BS triggered by ePVM processes via the GCM adapter.

5.4 Application Monitoring

Monitoring is one of the core features of every process engine and it is an important argument for using one when building an application. The ePVM engine supports monitoring processes by registering monitor objects for one or more process definitions. Furthermore, it can be specified which events shall be monitored. Available are a number of standard events such as a process instance being created, a message being processed, or a process becoming idle. Furthermore, custom events can be defined such that more fine-grained monitoring can be implemented, for example, multiple events can be triggered while a single message is processed.

In the BIS application, a monitor object is used to track the progress of ePVM process instances, for example, while the system initialization process is executed (c.f. Fig. 5). The monitor object is triggered by the process engine whenever activities start or finish and it updates the GUI to reflect the state of the system. Furthermore, it is desired to monitor the GCM component system with the goal to visualize AM actions and the number of workers used in the DP skeleton. A system administrator could observe this and, if required, trigger reconfiguration or add resources manually. For monitoring the skeleton, functionality provided by the GCM framework can be used. However, monitoring in GCM is based on a pull model where information about components and their states can be retrieved on

request. On the contrary, the ePVM approach can be considered a push model where a monitor is registered and receives events. To integrate GCM monitoring with the event-driven paradigm applied in ePVM some adaptation is necessary. A first solution is to create a dedicated ePVM process which regularly retrieves information about the component system via the GCM API and creates events for the monitor object. A second solution is to instrument the component implementation to actively send events to an ePVM process. The first approach is more generic with respect to distribution, as the GCM framework handles remote method invocations required to query for component states automatically. The second approach is more efficient, as communication only takes place if an event to be monitored occurs. However, a component might not be able to easily communicate with the process engine if it is running on a remote machine, since the process engine itself is not a GCM component. In the BIS we used the first approach to implement monitoring the number of workers, as the workers are typically distributed. For monitoring AM actions, we use the second approach exploiting the fact that in our deployments the AM is always co-located with the process engine such that no remote communication is necessary.

In general, the requirement to monitor actions within the DP skeleton to some extend is contradictory to the idea of autonomic components. On one hand the goal of using the DP skeleton is to take advantage of its built-in functionality without taking care of the implementation details. On the other hand, we still want to be able to monitor certain internal details such as reconfiguration operations and the number of workers. From the perspective of the process-driven applications paradigm all important actions which shall be monitored should be centrally controlled by the process-engine. However, in real-world applications a trade-off between central process control and autonomy must be made.

5.5 Automatic Futures vs. Message Passing

When integrating process-engines and distributed computing frameworks, it is very important to be aware of their communication and synchronization paradigms. The GCM framework is based on Java RMI and implements the concept of automatic futures (Caromel & Henrio 2005). This means that method invocations always return immediately, whereas results which are not yet available are represented by so-called future objects. Program execution is then blocked automatically if a future object is being accessed as long as the value represented is not yet available. The goal is to ease parallel programming by hiding synchronization details within a meta object protocol implemented in GCM. The ePVM process engine, however, uses message passing for communication and synchronization between concurrent control flows. If these two paradigms are interweaved, as it is the case in the BIS application, process flows can easily become distorted. For example, if a process definition assigns two activities to be carried out sequentially (c.f. activity 2 and 3 in Fig. 5), it must be ensured that no more future objects resulting from the first activity exist before the second is triggered.

This issue becomes obvious when an identification process is triggered within the BIS. In this case, an ePVM process sends a message to the GCM adapter including fingerprints of a person to be identified. The GCM adapter forwards this information to the component system by invoking the broadcast interface of

the DP skeleton (port S, Fig. 3). This interface is a so-called collective interface, which turns one method invocation into N method invocations on all the bound IDMatcher components to broadcast the identification request. The return value is a list of result objects, one from each IDMatcher component. When the interface is invoked, it immediately returns a list of future objects, which at the beginning are all unavailable and then by-and-by become available as the IDMatcher components return their results. It is important that the GCM adapter waits for the futures to become available and generates messages to be returned to the ePVM process instance accordingly. It must not report the identification as completed before all futures are available. Effectively, the GCM adapter retracts automatic synchronization in order to make the actual progress visible to the process engine, which must to be informed whenever an IDMatcher component has searched its part of the DB. Obviously, converting from one paradigm into the other must be handled with care as the semantics of the process definitions can be broken due to delayed synchronization within GCM.

6 Results, Experiences, and Lessons Learned

The primary result of this work is the fully functional prototype of the BIS application, which acts as a use case demo for the process engine as well as for the GCM framework. Additional results have been gained by critically evaluating the application and experimenting with it. Firstly, it has been successfully deployed on various hardware platforms ranging from one multi-core PC to heterogeneous sets of clusters as provided by the Grid5000 project (The Grid5000 Project 2008). Switching hardware platforms did not require changing a single line of functional code, only the infrastructure part of the XML deployment descriptor required modification. The strict separation of concerns and the autonomic functionality implemented within the GCM framework have turned out to be the main factors leading to this flexibility. The former ensures that resources are never directly referenced in the source code while the latter provides autonomic adaptation to the performance properties of the hardware in use.

Secondly, functionality and autonomic behaviour of the application has been verified using Grid5000. The BIS has been started using 50 workers (one per node), a DB holding 50000 identities (approx. 400 MB), and a QoS contract mandating a partition size of 1000 identities/worker. At runtime, the contract has been updated to 800 ($\pm$ 10%) identities/worker. Thereupon, the AM has successfully detected 7 contract violations and each time reconfigured the DP skeleton by adding one additional worker until a partition size of 877 identities/worker was reached at 57 workers/nodes. During this experiment, every reconfiguration operation took about 9 seconds in which the complete DB has been redistributed (from the node hosting the whole database to the nodes hosting the workers of the data parallel BS) by the ABC. When identification requests where issued during reconfiguration, they where queued automatically by the skeleton and processed as soon as reconfiguration was completed. For the given DB size, each identification request required around 10 seconds to be processed. This means that each reconfiguration operation roughly decreases the throughput of the BIS by one identification for any given timeframe. Therefore, if the BIS is used in a very dynamic environment requiring frequent

reconfiguration, the number of occurrences of reconfigurations may be sensibly reduced by adopting more aggressive parallelism degree variation policies, in such a way the overall overhead is reduced. Such more aggressive policies at the moment consist in varying the constant Δ_w that defines the number of workers to be added/removed when reconfiguring the parallelism degree of a BS. As shown in (Aldinucci et al. 2009*c*), the evaluation of such Δ_w can also take in account the overhead and the delay of the reconfiguration itself by using historical data. In the BS/GCM framework we are currently investigating the possibility to use a kind of exponential back-off increase/decrease protocol. All those cases, of course, rely on the possibility to effectively monitor the increase/decrease achieved in the BS performance as a consequence of the parallelism degree adaptation.

Finally, evaluating the application's source code, including the deployment descriptor required to run on 50 nodes of Grid5000, unveiled the source code breakdown illustrated in Fig. 6. The functional code mainly includes the host processes (c.f. Fig. 4) providing DB access, the GUI functionality, and the interfacing to the GCM components. Its absolute size is about 2500 lines of code, which is very small considering the the overall functionality provided by the application. This is due to the fact that the GCM framework provides all the functionality for distribution and autonomicity. Implementing this functionality from scratch not using GCM would have been significantly more effort. In particular, adding autonomic control to an application is virtually effortless if a matching behavioural skeleton is available. Only the QoS contract must be provided and a few non-functional interfaces used by the controller must be implemented within the worker component. In case of the BIS application, only about 200 lines of code where necessary for that. Furthermore, it is to be noted that more than a quarter of the source code (27%) consists of code interpreted at runtime. This code, including the deployment descriptor, the process definitions, the QoS contract, and the GCM component definitions, contains the main control logic and infrastructure

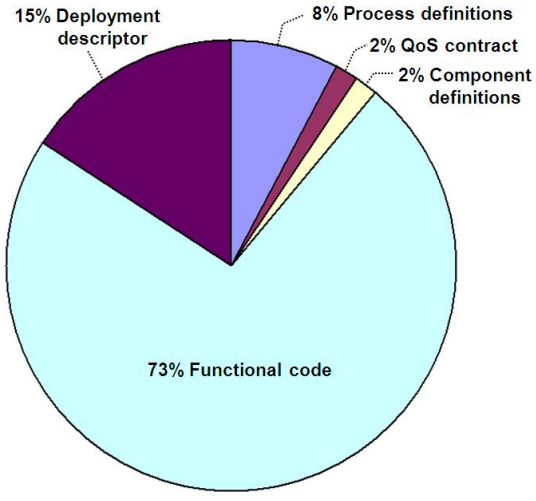

Figure 6 BIS source code breakdown

definition of the application. As a result, the application can be adapted significantly without recompilation - a very important property required for operation in today's dynamic business environments. Hard-coding this part of the application would clearly decrease the applications flexibility as achieved through the combination of GCM and ePVM.

During application development, we have made a number of experiences with regards to the integration of process technology and the GCM framework. The interfacing between the two technologies went rather smoothly, since the ePVM engine is available as a Java library and it does not dictate the use of Web services. Also, the DP skeleton fits well to the given biometric identification problem. However, application monitoring turned out to be challenging. One must be aware that the idea behind components is hiding complexity and this can be a problem if component internals need to be monitored. The GCM framework supports querying the state of a component system, however, it does not support monitoring activities within components, for example, reconfiguration within a BS. Solving this problem by instrumenting component implementations (c.f. Sec. 5.4) requires comprehensive knowledge of the GCM framework. Furthermore, the monitoring support of GCM follows a pull model while process engines are mostly event driven. Joining the two paradigms in a sensible way requires an extra effort and can have a performance impact. For example, regularly traversing component hierarchies to detect newly created components is not very efficient.

Another lesson we have learned is that the two different synchronization paradigms applied in GCM and ePVM can interfere if not handled with care. The concept of automatic futures implemented in the GCM framework follows the wait-by-necessity idea. This means that unavailable results are replaced by future objects such that synchronization is delayed as long as possible. Therefore, it must be carefully checked if results of activities within a process flow include one or more future objects before the next activity of a sequence is triggered, otherwise the process semantics can easily become distorted. In other words, if a GCM component returns an object it does not necessarily mean that all the related operations have completed.

Finally, we realized that working with the advanced features of both frameworks, ePVM and GCM, requires working with a large number of different development artefacts and acquiring related skills. The Grid IDE (GIDE) (Basukoski et al. 2008), which consists of a set of plugins to the famous Eclipse development environment eases this to some extend and provides a jump start into GCM. Nevertheless, combining process technology with GCM allows producing extremely flexible and complex distributed applications with minimum effort.

7 Conclusions

Process-driven application development is increasingly gaining attention in the business environment. At the same time, software development frameworks for the Grid/Cloud are raising interest in the course of the Cloud computing wave. In this paper we have considered combining the two approaches to produce a process-driven distributed biometric identification system. In discussing the application we have made the following contributions:

- We provided a brief overview of the GCM framework, its support for autonomic components and behavioural skeletons, and the ePVM process engine.
- We described the architectural design and implementation of the process-driven biometric identification system utilizing the DP autonomic behavioural skeleton available in GCM.
- We presented the results, experiences, and lessons learned while integrating both technologies, the process engine and the GCM framework.

We believe that this use case application demonstrates that combining process technology and autonomic Grid/Cloud components represents a powerful approach for developing flexible distributed applications with minimum effort. Obviously, the application could have been developed without using GCM and ePVM. However, the development effort would have been much higher and the resulting application would have been less flexible due to the hard-coded application logic and autonomic strategy.

References

Aldinucci, M. & Danelutto, M. (2006), 'Algorithmic skeletons meeting grids', *Parallel Computing* **32**(7), 449–462.

Aldinucci, M., Campa, S., Danelutto, M., Dazzi, P., Kilpatrick, P., Laforenza, D. & Tonellotto, N. (2007), Behavioural skeletons for component autonomic management on grids, *in* 'CoreGRID Workshop on Grid Programming Model, Grid and P2P Systems Architecture, Grid Systems, Tools and Environments', Heraklion, Crete, Greece.

Aldinucci, M., Danelutto, M. & Kilpatrick, P. (2009*a*), Autonomic management of non-functional concerns in distributed and parallel application programming, *in* 'Proc. of Intl. Parallel & Distributed Processing Symposium (IPDPS)', IEEE, Rome, Italy, pp. 1–12.

Aldinucci, M., Danelutto, M. & Kilpatrick, P. (2009*b*), Towards hierarchical management of autonomic components: a case study, *in* F. S. Didier El Baz, Tom Gross, ed., 'Proc. of Intl. Euromicro PDP 2009: Parallel Distributed and network-based Processing', IEEE, Weimar, Germany, pp. 3–10.

Armstrong, R., Gannon, D., Geist, A., Keahey, K., Kohn, S., McInnes, L., Parker, S. & Smolinski, B. (1999), Toward a common component architecture for high performance scientific computing, *in* 'Proc. of the 8th Intl. Symposium on High Performance Distributed Computing (HPDC'99)'.

Basukoski, A., Getov, V., Thiyagalingam, J. & Isaiadis, S. (2008), Component-based development environment for grid systems: Design and implementation, *in* M. Danelutto, P. Frangopoulou & V. Getov, eds, 'Making Grids Work', CoreGRID, Springer, chapter Component Programming Models, pp. 119–128.

Bouziane, H., Pérez, C. & Priol, T. (2008), A Software Component Model with Spatial and Temporal Compositions for Grid Infrastructures, *in* 'Proc. of the 14th Intl. Euro-Par Conference', Vol. 5168 of *LNCS*, Springer, Las Palmas de Gran Canaria, Spain, pp. 698–708.

Bruneton, E., Coupaye, T. & Stefani, J.-B. (2003), *The Fractal Component Model, Technical Specification*, ObjectWeb Consortium.

Bukovics, B. (2007), *Pro WF: Windows Workflow in .NET 3.0*, Apress.

Caromel, D. & Henrio, L. (2005), *A Theory of Distributed Object*, Springer-Verlag.

Cole, M. (2004), 'Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming', *Parallel Computing* **30**(3), 389–406.

CoreGRID project (2007), *Deliverable D.PM.04 – Basic Features of the Grid Component Model (assessed)*, CoreGRID NoE deliverable series, Institute on Programming Model. `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

Danelutto, M. (2005), QoS in parallel programming through application managers, *in* 'Proc. of Intl. Euromicro PDP: Parallel Distributed and network-based Processing', IEEE, Lugano, Switzerland, pp. 282–289.

David, P.-C. & Ledoux, T. (2006), An aspect-oriented approach for developing self-adaptive fractal components, *in* W. Löwe & M. Südholt, eds, 'Proc. of the 5th Intl Symposium Software on Composition (SC 2006)', Vol. 4089 of *LNCS*, Springer, Vienna, Austria, pp. 82–97.

Dowling, J. (2004), The Decentralised Systems Coordination of Self-Adaptive Components for Autonomic Computing Systems, PhD thesis, University of Dublin, Trinity College.

Faura, M. V. & Baeyens, T. (2007), *The Process Virtual Machine*. `http://www.onjava.com/pub/a/onjava/2007/05/07/the-process-virtual-machine.html`.

Fox, G. C. & Gannon, D. (2006), 'Special issue: Workflow in grid systems: Editorials', *Concurr. Comput. : Pract. Exper.* **18**(10), 1009–1019.

Freeman, S. & Pryce, N. (2006), Evolving an embedded domain-specific language in java, *in* 'Proc. of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)', pp. 855–865.

GridCOMP project (2008), 'Grid Programming with Components, An Advanced Component Platform for an Effective Invisible Grid'. `http://gridcomp.ercim.org`.

Institute on Programming model of the CoreGRID NoE (2009), *Home page*, CoreGRID NoE. `http://www.coregrid.net/mambo/content/blogcategory/13/292/`.

Kephart, J. O. & Chess, D. M. (2003), 'The vision of autonomic computing', *IEEE Computer* **36**(1), 41–50.

Parashar, M., Liu, H., Li, Z., Matossian, V., Schmidt, C., Zhang, G. & Hariri, S. (2006), 'AutoMate: Enabling autonomic applications on the Grid', *Cluster Computing* **9**(2), 161–174.

Ryan, K. L. K. (2009), 'A computer scientist's introductory guide to business process management (bpm)', *ACM Crossroads* **15**(4), 11–18.

Smith, H. & Fingar, P. (2006), *Business Process Management: The Third Wave*, Meghan-Kiffer Press, ISBN 0929652347.

The Grid5000 Project (2008), 'An infrastructure distributed in 9 sites around France, for research in large-scale parallel and distributed systems'. `http://www.grid5000.fr`.

Vanneschi, M. (2002), 'The programming model of ASSIST, an environment for parallel and distributed portable applications', *Parallel Computing* **28**(12), 1709–1732.

Weigold, T., Kramp, T. & Buhler, P. (2007), ePVM - an embeddable Process Virtual Machine, *in* 'Proc. of the 31st Intl. Computer Software and Applications Conference (COMPSAC)', Beijing, China, pp. 557–564.

zur Muehlen, M. (2004), *Workflow-based Process Controlling - Foundation, Design, and Application of Workflow-driven Process Information Systems*, Logos Verlag, ISBN 3-8325-0388-9.