Chapter 2

Existing Parallel and Distributed Systems, Challenges, and Solutions

2.1 Modern Software Architectures, Systems, and APIs for Parallel and Distributed Systems

This chapter presents the most important of modern distributed software architectures by outlining key components and the way the latter interact and synchronize. Correspondingly, examples of representative implementations of these architectures are given with discussions on the APIs available to the client. Secondly, workflow management systems for various environments are shown as a feasible way of integrating software components within particular types of systems. This is done as a prerequisite to analysis and discussion of future directions, challenges in *dynamic* integration of *various* types of distributed software to which the author proposes solutions covered in depth further in this book.

2.1.1 HPC Systems

High-performance computing [46, 47] has come a long way from dedicated, extremely expensive supercomputers to machines equipped with multicore CPUs and powerful GPU devices widely available even in our homes today. As an example, Intel Xeon Processor E7-8890 v2 features 15 cores, Intel Xeon Phi 7120 features 61 cores. NVIDIA GeForce GTX Titan features 2688 CUDA cores. AMD Opteron 6380 Series Processors feature 16 cores. Figures 2.1 and 2.2 present the performance in Tflop/s and the number of cores of the first cluster on the TOP500 [19] list over the last years. Figure 2.3 shows the average processor frequency used in the ten most powerful clusters on the TOP500 list in respective years. It can be seen that the growth in the clock frequency has practically stopped and consequently the increase in the performance is mainly due to a growing number of processor cores in the cluster as well as adoption of new computing devices. Apparently, this will also be the tendency of cheaper machines which can accommodate GPU devices easily assuming a powerful enough power supply is installed.

Adoption of more computational devices (such as computers, cluster nodes but also cores within devices) has crucial consequences for software. In order to use the available processing capabilities, one or more of the following techniques needs to be adopted:

1. launching more applications with processes/threads to use the available processors and cores,

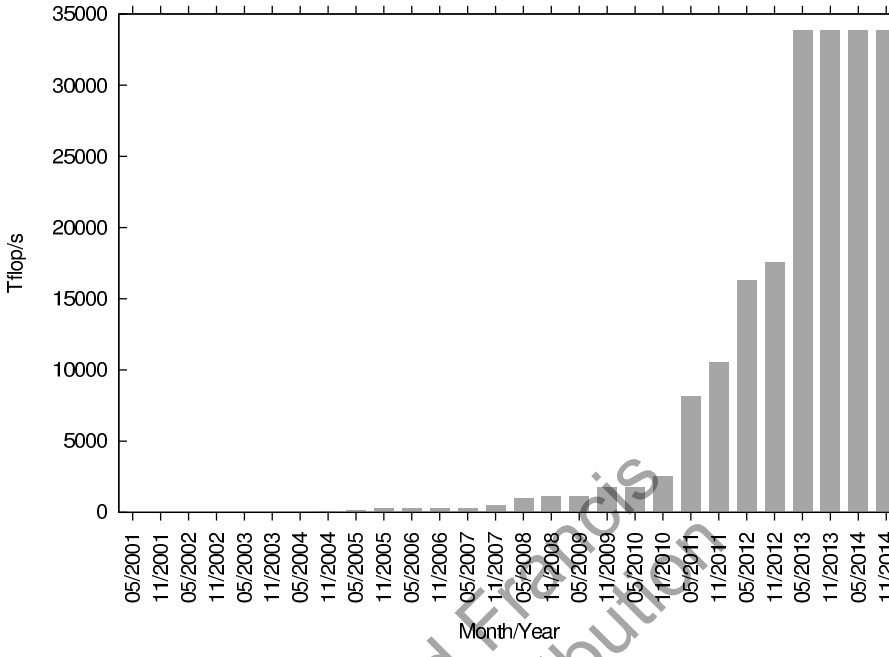

FIGURE 2.1: Performance of the first cluster on the TOP500 list, based on data from [19].

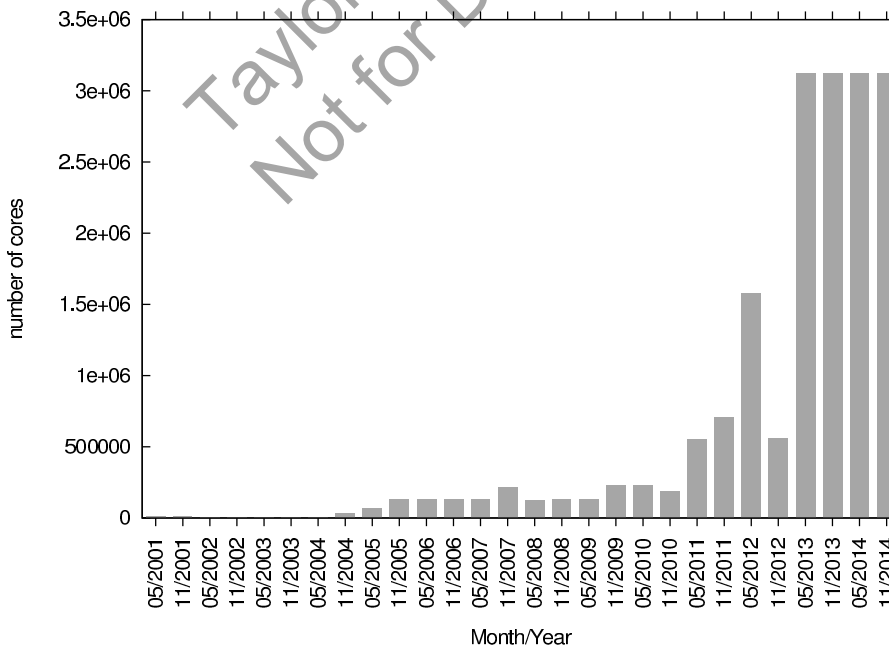

FIGURE 2.2: Number of processing cores of the first cluster on the TOP500 list, based on data from [19].

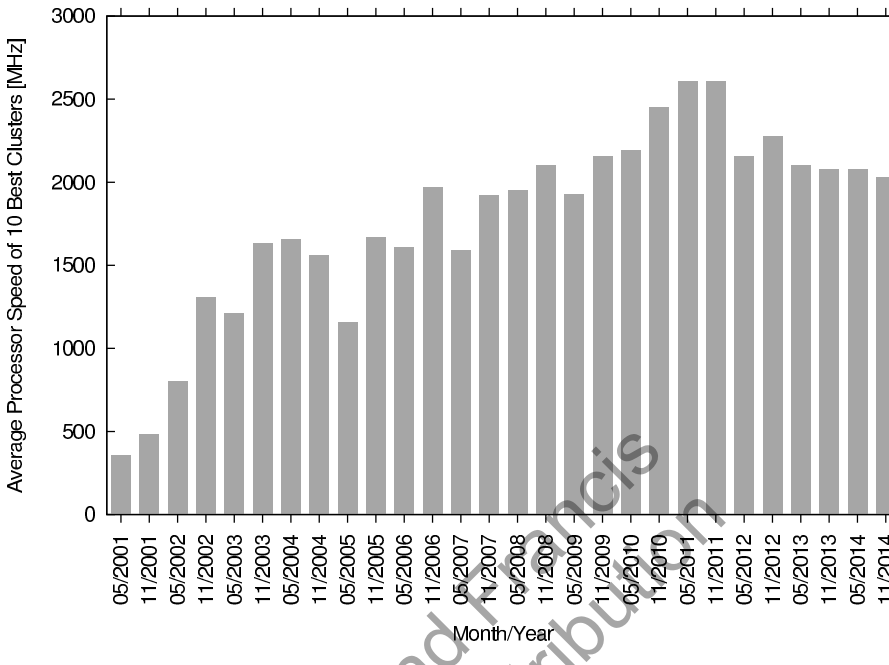

FIGURE 2.3: Average processor speed of the ten best clusters on the TOP500 list, based on data from [19].

2. parallelize existing applications so that these launch more threads to make use of the large numbers of cores.

In order to do this, a proper API and runtime systems are needed. The most popular ones are discussed next along with access and management of the parallel software.

2.1.1.1 GPGPU and Accelerators

NVIDIA CUDA [129, 188, 161] and OpenCL [38] allow General-Purpose computing on Graphics Processor Units (GPGPU). NVIDIA CUDA offers an API that allows programming GPU devices in the SIMT fashion. The following concepts are distinguished regarding the application paradigm:

1. a 3D grid that is composed of blocks aligned in three dimensions X, Y, and Z, in which
2. each block consists of threads aligned in dimensions X, Y, and Z.

The usual steps in application execution include the following:

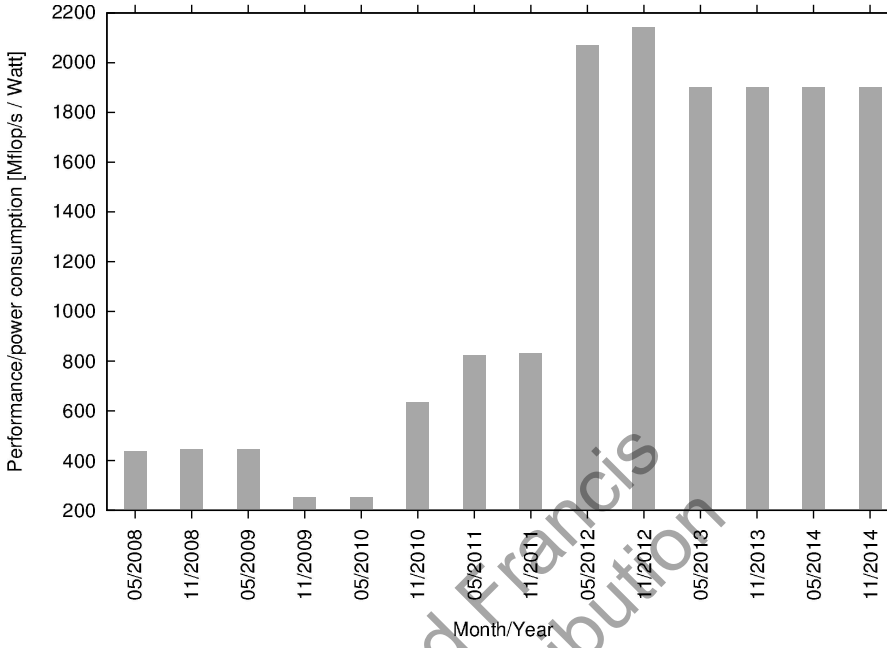

FIGURE 2.4: Performance per Watt of the first cluster on the TOP500 list, based on data from [19].

1. discovery of available GPU devices;
2. uploading input data from the RAM memory of the computer into the global memory of the device;
3. specification of grid sizes, both in terms of block counts in three dimensions as well as thread counts in three dimensions within a block;
4. launching a kernel, which is a function that is executed in parallel by various threads that process data; and
5. downloading output data produced by the kernel from the global memory of the GPU to the RAM.

Advanced code will use several optimization techniques such as:

- data prefetching along with usage of faster but smaller shared memory as well as registers;
- memory coalescing, which speeds up access to global memory;

- optimization of shared memory and register usage, which may increase the parallelization level;
- loop unrolling;
- launching several kernels and overlapping communication and computations;
- using multiple GPUs.

OpenCL offers an API similar to that of NVIDIA CUDA but generalized to run not only on GPUs but also on multicore CPUs. This allows us to use a modern workstation with several multicore CPUs and one or possibly more GPUs as a cluster of processing cores for highly parallel multithreaded codes. Because of that, OpenCL requires some more management code related to device discovery and handling but the kernel and grid concepts have remained analogous to NVIDIA CUDA.

Development and execution of both NVIDIA CUDA- and OpenCL-based applications involve the following:

1. logging into the computer equipped with CPU(s) and GPU(s), e.g., using SSH,
2. compilation of the code using a proper compiler, e.g., `nvcc` for NVIDIA CUDA and `nvcc -lOpenCL` for OpenCL programs, and
3. execution of the program.

As explained above, the aforementioned solutions require access to HPC resources using, e.g., SSH as well as programming and operating system knowledge to develop and run corresponding applications.

OpenACC [178, 217], on the other hand, allows us to extend sequential code using directives and functions that allow subsequent parallelization of the application when running on GPUs. This approach is aimed at making parallel programming on GPUs easier similarly to parallelization using OpenMP [210, 49].

Intel Xeon Phi coprocessors allow us to use one of a few programming APIs such as OpenMP [210, 49], OpenCL [38] or Message Passing Interface (MPI) [36] described below.

2.1.1.2 Clusters

Traditionally, the Message Passing Interface (MPI) [36, 218] has been used for multiprocess, multithreaded programming on clusters of machines, each of which may use multiple, multicore CPUs. MPI is a standard API available to a parallel application that consists of many processes that work on separate processing units (possibly on different nodes of a cluster). Processes interact with each other by exchanging messages. Moreover, a process can include several

threads that are allowed to invoke MPI functions provided the MPI implementation supports one of the multithreading levels. This, apart from MPI non-blocking functions, allows overlapping computations and communication. Furthermore, important features of MPI include:

- potentially hardware-optimized collective communication routines,
- parallel I/O allowing several processes to access and modify a file(s),
- dynamic spawning of processes,
- partitioning processes into groups and using various communicators,
- additional functions of certain MPI implementations such as transparent checkpointing in LAM/MPI and BLCR [214].

Development of MPI programs, access and management usually involves the following steps (after logging into the HPC system via SSH):

1. development of source code (e.g., in C/C++ or Fortran),
2. compilation using a compiler with MPI libraries (such as using `mpicc`),
3. submission into a queuing system such as Portable Batch System (PBS), Load Sharing Facility (LSF), or LoadLeveler [141] or running from the console,
4. checking status and browsing results of the application which can be done using the respective commands of queuing systems.

There exist higher-level graphical interfaces that hide details of application submission, checking status and browsing results. Usually these are interfaces built on top of grid middleware able to access several HPC clusters as described in Section 2.1.5. Examples of such interfaces include BeesyCluster, co-developed by the author [69], or MigratingDesktop [12, 136], developed within the CrossGrid project.

MPI programs can be combined with APIs for shared memory programming such as OpenMP [112]. Furthermore, in clusters in which nodes are equipped with GPUs, MPI can be used together with, e.g., NVIDIA CUDA in order to exploit all parallelization levels. When nodes have Intel Xeon Phi coprocessors installed, a single MPI application can be launched in such a way that some processes run on CPU cores and others on cores of coprocessors. There are also other solutions that can help use the available resources in cluster environments. For instance, rCUDA [17, 86] allows sharing GPUs among nodes of a cluster in a transparent way. Many GPUs Package (MGP) [39] through implementation of the OpenCL specification and OpenMP extensions enable a program to use GPUs and CPUs in a cluster. KernelHive [79] allows us to define, optimize and run a parallel application with OpenCL kernels on a cluster with CPUs and GPUs. It is possible to use various optimizers such as minimization of execution time and a bound on power consumption of devices selected for computations.

2.1.2 SOA

Service Oriented Architecture (SOA) [88] assumes that a distributed application can be built form distributed *service*s. The service is a software component that can be developed by any provider and made available from a certain location through well-known interfaces and protocols. Such loosely coupled services can be integrated into a complex application that uses various services to accomplish particular tasks.

The popular Web Service technology can be viewed as an implementation of SOA with the following key standards and protocols [185]:

1. SOAP [212]—a protocol that uses XML for representation of requests and responses between the client and the service.
2. WSDL (Web Service Description Language) [213]—a language used to represent a technical description of a Web Service. It contains operations available within the service with specifications of input and output messages, used data types, the location of a service and protocol binding.
3. UDDI (Universal Description, Discovery and Integration) [21]—a standard aimed at development of service registries that could be used by the community. It allows publishing data in a secure way and querying the registry for business entities or services published by them. Providers and services can be described by means of classification systems as well as contain technical characteristics or links to WSDL documents.

The technology features:

- *Self-contained services*—each service is published by its provider and its code is contained within the service and possibly other services it may use. The client does not need any other software other than for encoding a request so that it is accepted by the service and decoding its result,
- *Loosely coupled services*—services are published independently by their providers and are *interoperable* thanks to SOAP.

The standard sequence that allows us to consume a Web Service involves the following steps:

1. discovery of a service that performs a desired business function,
2. fetching the description of the service,
3. building a client and invoking the service.

Figure 2.5 presents a sequence which has been extended compared to the one cited in the literature [185] by inclusion and application of additional standards and protocols to improve the discovery process with semantic search. Semantic computing [196] is gaining much attention in distributed information systems. The steps using these standards are as follows:

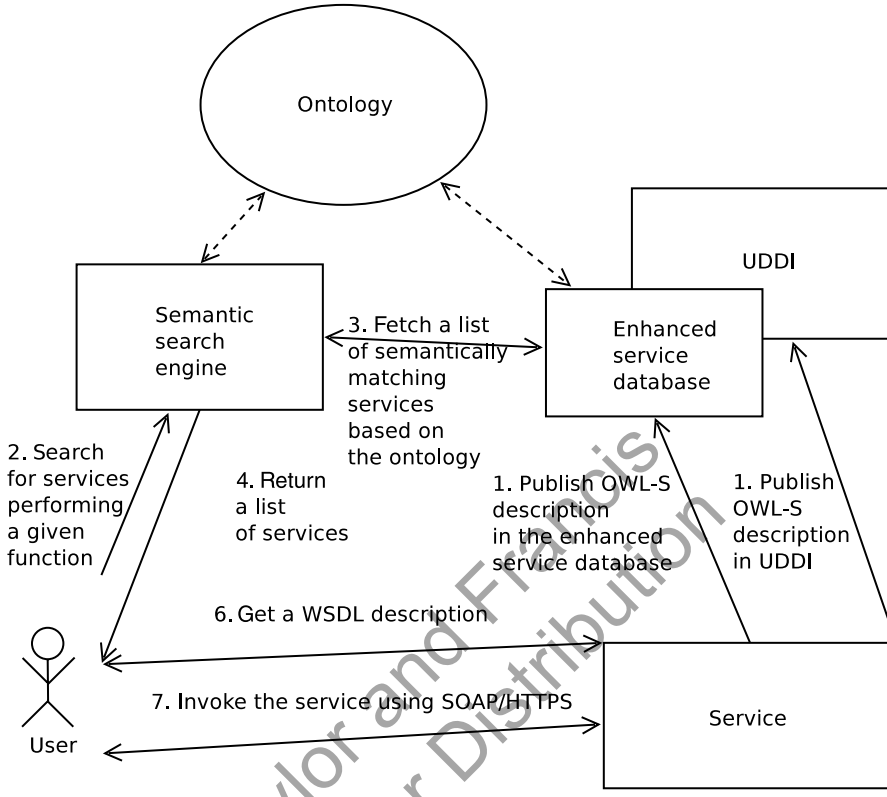

FIGURE 2.5: Publishing, semantic search for services, and service invocation sequence.

1. Publishing information about the service using the OWL-S [82]. OWL-S allows us to specify:

   - Inputs,
   - Outputs,
   - Preconditions,
   - Results.

   Proper concepts/OWL-classes can be used in these fields. Next, the OWL-S description can be mapped into an UDDI description [198].

2. The client sends a request to a UDDI server [185]. In the traditional approach, it is the UDDI server that will search for the service among its records. However, searching in UDDI has been limited to matching keywords with no semantic relation between concepts present in the

search request and the description in the registry. Thus, instead of the UDDI server, the request can be routed to a more advanced semantic engine that will consider an ontology for the search. Namely, for each of the concepts on a particular field of the specification of the request, the engine tries to match and find semantic links to the concepts of the corresponding field of the service description. Paper [198] describes how such semantic connections can be determined using inheritance of concepts.

3. A list of services matching the client's request is returned. The client selects a service and fetches its description in WSDL.

4. The client invokes the service using SOAP, which is transferred using one of many possible protocols, the most common being HTTP and HTTPS.

5. After the service has processed the request, a response wrapped into SOAP is sent back to the client.

The RESTful service [111] solution is an alternative to SOAP-based services. REST (REpresentational State Transfer) [92] distinguishes resources identified by URIs and uses HTTP methods to manipulate the resources. As such, REST can be thought of as a technology allowing realization of Resource Oriented Architecture (ROA) [145]. ROA distinguishes the concept of a resource that is accessible through certain interfaces. Such services can be described by either WSDL 2.0 [151, 213] or Web Application Description Language (WADL) [110]. Technically, REST can also be thought of as an implementation of a (part of) SOA [89].

What is crucial is the possibility of invoking not only one service but to combine outputs of several services and process these further in order to produce output of a given service. This can also be done at more levels of such a service invocation hierarchy. For instance, a travel agency could publish a service for booking a package holiday. After placing a request by invocation of the service, the service would need to find out what other services are available out there to perform more concrete tasks such as:

- booking a flight,
- booking hotels,
- booking additional activities, attractions, events,
- checking weather forecasts for particular places.

Note that there may be several providers offering a particular service, e.g., booking a flight, in which case there will be several alternative services that perform the requested function. Such services can differ in quality metrics such as price, time to book, reservation period, reliability, conditions to cancel, etc.

It can be noted that it may be the travel agency that selects the best flight out of those offered by various airlines or it can further delegate this task by invoking a service of another company that specializes in this very process and does that better for a reasonable fee. Proper selection of such services and execution of such a complex scenario is discussed in depth in Section 2.2.

2.1.3 Multitier Architectures

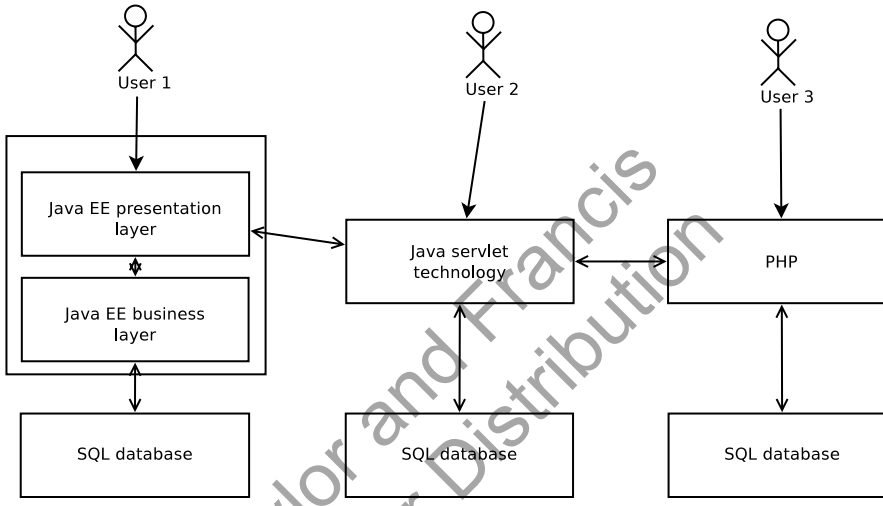

FIGURE 2.6: Multitier architecture.

The multitier approach in distributed system design distinguishes multiple, functionally distinct layers that can be integrated into an application. This has several advantages such as:

- independent development of layers,
- possibility of independent modification,
- possibility of invoking a certain layer from another application.

It is important to note that each layer of such a system exposes its own API to its clients. Typical layers include:

1. The client can be a thin client (Web browser), application running in a sandbox (such as a Java applet), or a desktop application.
2. Server layer that accepts user input, processes it, and sends results back. In some technologies it can be further broken into more layers such as:

- presentation—for verification, parsing input data passed by the client of this layer and presentation of results,
- business—for actual processing of the request and performing the requested function; this layer might contact other layers or other cooperating systems to do its job.

3. Database for storage and retrieval of data. The database engine can aggregate and return only results requested by the client of this layer.

Figure 2.6 depicts three multitier systems with popular technologies and popular APIs exposed by the layers.

2.1.4 Peer-to-Peer, Distributed Software Objects, and Agents

Peer-to-peer computing is a concept of distributed computing in which distributed components cooperate by being able to serve as providers and clients at the same time. In principle, the components are equal and can divide particular tasks of the scenario and management of data among themselves.

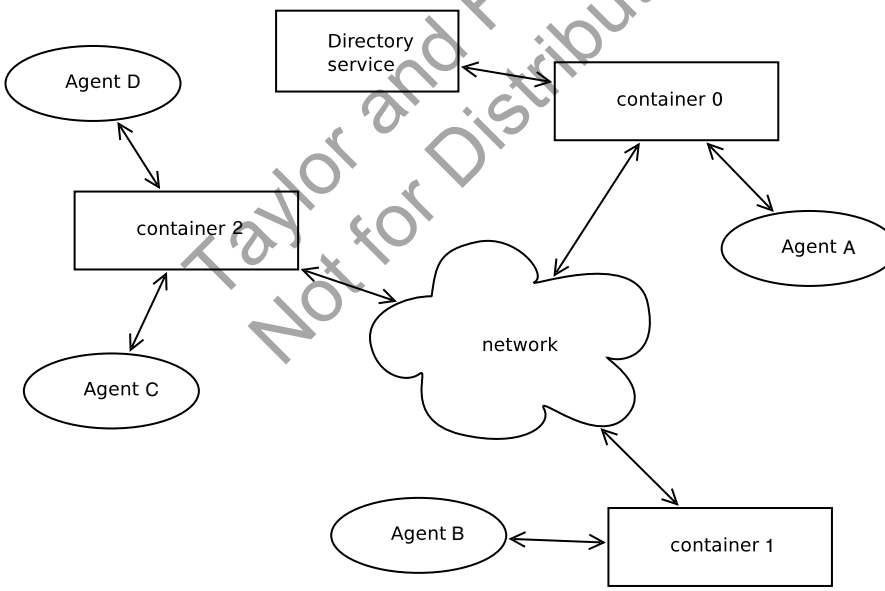

FIGURE 2.7: Distributed agents architecture.

The distributed software objects approach distinguishes distributed objects as components that can interact with each other by invoking exposed methods that operate internally on object attributes. An example of such

type of computing is CORBA (Common Object Request Broker Architecture) [194, 115], which allows communication between distributed objects using an Object Request Broker (ORB) [164]. Objects can be implemented in various languages and run on one of many operating systems, in particular using:

- VBOrb [23] for Visual Basic,
- MICO [175] for C++,
- omniORB [13] for C++ and Python,
- ORBit2 [15] for C and Perl, or
- IIOP.NET [8] implementation of IIOP for the .NET environment.

The publish and invocation sequence involves the following steps:

1. Specification of an object interface using the OMG Interface Definition Language (IDL).
2. Translation of the interface to a skeleton on the server side or a stub on the client side. These codes are generated in the language the corresponding side will be implemented.
3. Augmenting the code with actual service implementation on the server side or service invocation on the client side.
4. Creation of objects and storing and passing of Interoperable Object References (IORs) to the client or object registration in the Naming Service (see below).
5. Calling object methods by other objects.

Objects can use services [93] such as the Naming Service, Security Service, Event Service, Persistent Object Service, Life Cycle Service, Concurrency Control Service, Externalization Service, Relationship Service, Transaction Service, Query Service, Licensing Service, Property Service, Time Service, Trading Service, and Collection Service.

Distributed Software Agents [42] can be treated as further extension of distributed objects and be applied on top of the peer-to-peer processing [155]. Particular agents are instances of agent classes and can communicate with each other by sending messages. The following key features of multiagent systems can be distinguished:

- *Autonomous agents*—agents act independently and can perform own decisions,
- *Global intelligence but local views of the agents*—agents can cooperate to achieve a global goal but each agent has its own perception of the part of the environment it can observe. Together agents can acquire and use knowledge that would not have been available for a single party,

- *Self-organization*—cooperating agents can arrange parts of a complex task themselves; on the other hand, agents may need to look for new agents to fetch information, trade, etc., which requires:
  - *negotiation* with other agents,
  - *ontology* as a means of common understanding of used terms for communication between agents that do not know each other.
- *Migration*—agents can migrate throughout a distributed system to gather and use information; this approach might be more efficient than multiple client-server calls; this approach can also cope with partitioning of the system which would make calls from a distant client impossible.

Figure 2.7 presents the architecture of a distributed system which includes containers on which distributed agents work. The agents need a directory service to find and communicate with other agents. An example of such a platform is JADE [42] in which agents have defined behaviors and can communicate with each other by sending ACL messages. Each agent can have its own logic. Two special agents are available: AMS for enabling launching and control of agents and a Directory Facilitator (DF) that allows agents to advertise their services.

It should be noted that agents can be exposed to Web Service clients using the JADE Web Services Integration Gateway (WSIG) [118].

2.1.5 Grid Systems

Grid computing [97, 98, 96] can be best defined as controlled resource sharing. It was proposed as a means of integration of various resources so that there is a uniform way and API for using these resources irrespective of where the resources are, how they differ, or who has provided them. Theoretically, computing resources could be used much like suppliers of electric energy, i.e., connect to the system and use it with the system taking care of which resources are actually used for the request.

Grid computing specifies so-called Virtual Organizations (VOs) that are separate units in different administrative domains. The VOs have certain computing resources at their disposal that can use specific

1. operating systems,
2. storage systems,
3. user authentication and authorization mechanisms,
4. policies of how, when, and if the system can be accessed from outside; the latter can also differ on whether the user belongs to a particular group of users,
5. access protocols and APIs (e.g., WWW, SSH, etc.).

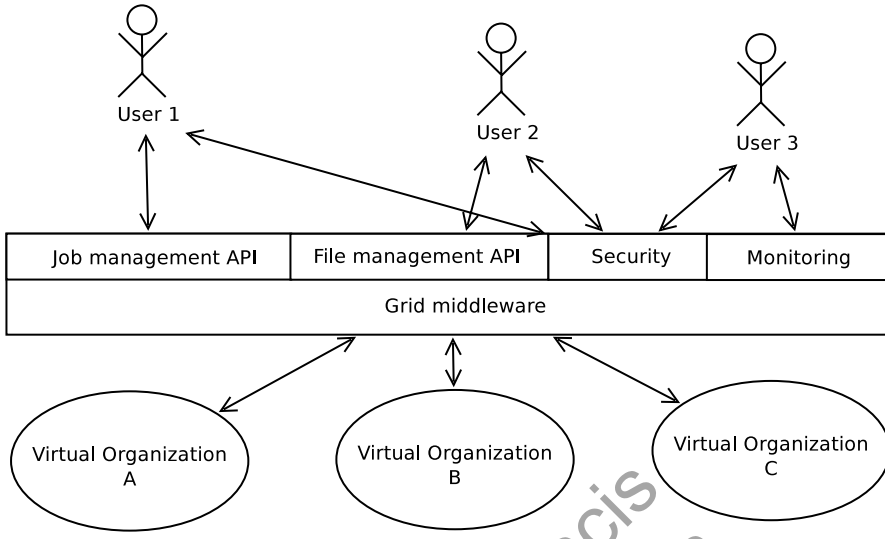

FIGURE 2.8: Interaction between the user and grid middleware and between grid middleware and virtual organizations.

The grid system couples resources of two or more Virtual Organizations into one system. This is done through a particular *grid middleware* that hides all the aforementioned differences among VOs and offers a uniform API to the client of the grid system. The client can use the API and the grid can take care of where the task is executed (Figure 2.8).

Actual implementations of grid systems such as Globus Toolkit [7, 197] offer APIs that allow the following:

1. execution of applications on the grid;
2. management of data on the grid including transfer of files to/from the grid and between particular grid locations; input file staging and output file staging, as well as cleanup after execution of a task;
3. security management including
   - obtaining security credentials to access the grid,
   - right delegation;
4. monitoring resources.

Globus Toolkit is probably the most popular grid middleware. Its version 5 features the following components:

- Grid Resource Allocation Manager (GRAM) [206] for location, submission, canceling and monitoring jobs on the grid by communication with various job schedulers.
- GridFTP [207] protocol for secure, reliable and high-performance data movement in WANs.
- Grid Security Infrastructure (GSI) [208] for authentication, authorization and management of certificates. It uses X.509 certificates and allows delegation through proxy certificates.
- Common Runtime.

The previous Globus Toolkit 4.x version implemented the Open Grid Services Architecture (OGSA) and Web Service Resource Framework (WSRF) [53, 197] and additionally provided a Monitoring and Discovery System (MDS).

Other available grid middleware include UNICORE [22, 35, 201], and gLite [87, 138].

On top of grid middleware, several high-level interfaces and systems have been developed that make running applications, management of data, and information services easier to use. These do allow us to run specialized parallel applications in the grid environment, the former often tailored to the needs of specific domain users. Examples include, among others, CrossGrid [105], CLUSTERIX [219, 131, 134], EuroGrid [139], and more recently PL-Grid [51, 43, 130] in Poland. These do require additional functionality compared to stateless Web Services as the client-server interaction often requires a context to be retained between successive calls for the identification of the client to return, e.g., results of previous requests. Furthermore, the client must be prevented from unauthorized access to resources they should not have access to. Accounting, secure data transmission and resource discovery are crucial for distributed computing between virtual organizations.

2.1.6 Volunteer Computing

Volunteer computing allows distributed execution of a project by a large group of geographically distributed user volunteers. Similar to grid computing, it is about integration of resources (especially computational) in a controlled way. However, there are also key differences. Table 2.1 lists both similarities as well as different solutions adopted in these approaches.

The simplest architecture of a distributed volunteer-based system can contain one project server with multiple distributed volunteers connected to it. The client computers perform the following steps in parallel:

1. Define usage details, i.e., CPU percentage/cores, RAM, and disk, that can be used by the system.
2. Download computational code from the project server.

TABLE 2.1: Grid vs volunteer computing.

Feature	Grid Computing	Volunteer Computing
Integration of resources in controlled way	+	+
Participating parties	Virtual Organizations (VOs)	End users
Large-scale distributed processing	+	+
Suitable only for compute-intensive applications without dense communication	+	+
Uniform job submission and file management API for clients who want to use resources	+	—
Security and reliability of computations	Trusted VOs	End users cannot be trusted; need for replication
Social undertaking	—	+
Safe for the client	+ (Node code executed on the client side)	-/+ Needs setting up a sandbox such as a separate UNIX account for real safety

3. in a loop:
   (a) download a packet(s) of input data,
   (b) process the data,
   (c) report results and effort in terms of CPU and memory used over the processing period.

The aforementioned scheme is typical of volunteer-based systems such as BOINC [30]. The BOINC system allows running several independent projects that attract volunteers to donate processing power, memory and storage to process data packets. Management of such a project, keeping the volunteers active and attracting new ones becomes a socially oriented initiative. The BOINC server may need to send two or more packets of data to various clients to make sure that returned results are correct. This may allow us to assess reliability of particular clients over time and rely on single processing requests. Similarly, assessment of the reported effort is based on the minimum out of reported efforts from two volunteers.

Comcute [6, 37, 75] is a system for volunteer computing in which the server layer is extended compared to BOINC for purposes of reliability and security of computations. Contrary to BOINC, computations on the client side are

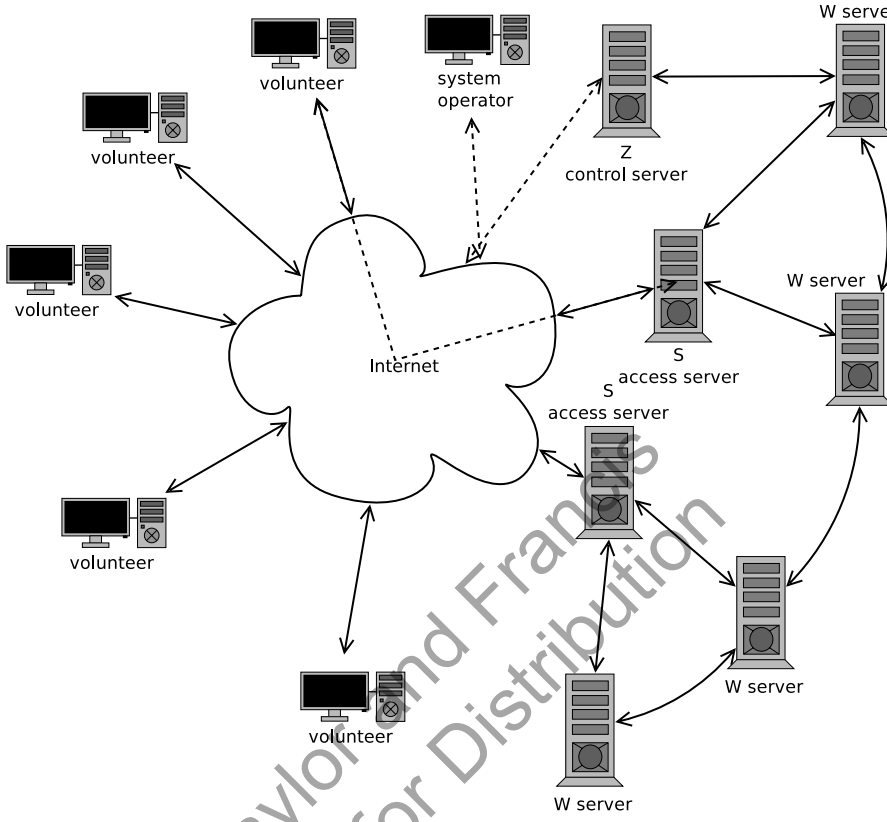

FIGURE 2.9: Volunteer computing architecture.

executed within a web browser, thus relieving the client from any installation of any application. Figure 2.9 presents the architecture of the system with the following layers:

1. Servers (W) for management of computations; within these servers a group of servers is elected for handling the task in a collective manner by distribution of data packets among S servers connected to the W layer.

2. Distribution servers (S) to which volunteers connect in order to:

   (a) Submit the capabilities of the client system, i.e., the computing technologies it supports, e.g., JavaScript, Java, Flash, Silverlight.

(b) Obtain computational code; the server can use the information provided in the previous step to provide the code in the most efficient technology.

(c) Successively fetch input data and return results.

3. Access server (Z) that exposes the access API and a Web interface as an indirect way to access W servers; for security reasons it is not possible to connect to W servers directly.

In Comcute it is possible to define the logical structure of W servers, connections between S and W servers, redundancy used when sending requests to particular volunteers, and the number of W servers in charge of a particular project. From the API point of view, the programmer needs to provide code for a data partitioner, computations and data merger.

2.1.7 Cloud Computing

Cloud computing [29] has become widespread and popular in recent years. The client is offered various kinds of services that are made available from *the cloud* and managed by providers. Be it infrastructure, particular software, or a complete hardware, operating system, and a software platform, all options are marketed as being:

1. *cheaper* than purchasing the necessary equipment and/or software licenses by the client, necessary maintenance, administration, introducing software updates;

2. *scalable* because the client can often scale the configuration not only up but also down depending on current needs;

3. *convenient and easy to use* as the client does not need to focus on hardware and software management if the core business is not related to computer science; in this scenario the client does not deal with any hardware failures, software incompatibilities, etc.;

4. *accessible* as the cloud can be accessed from practically anywhere using any device assuming the Internet is available.

On the other hand, the following should be noted:

1. *The data is stored in the cloud* and is handled by the provider and the client does not always know where and how the data is stored. Depending on where the provider offers their services and where the data is located, various laws may govern; consequently the legal aspects might not be clear for the client unless specified clearly in the agreement.

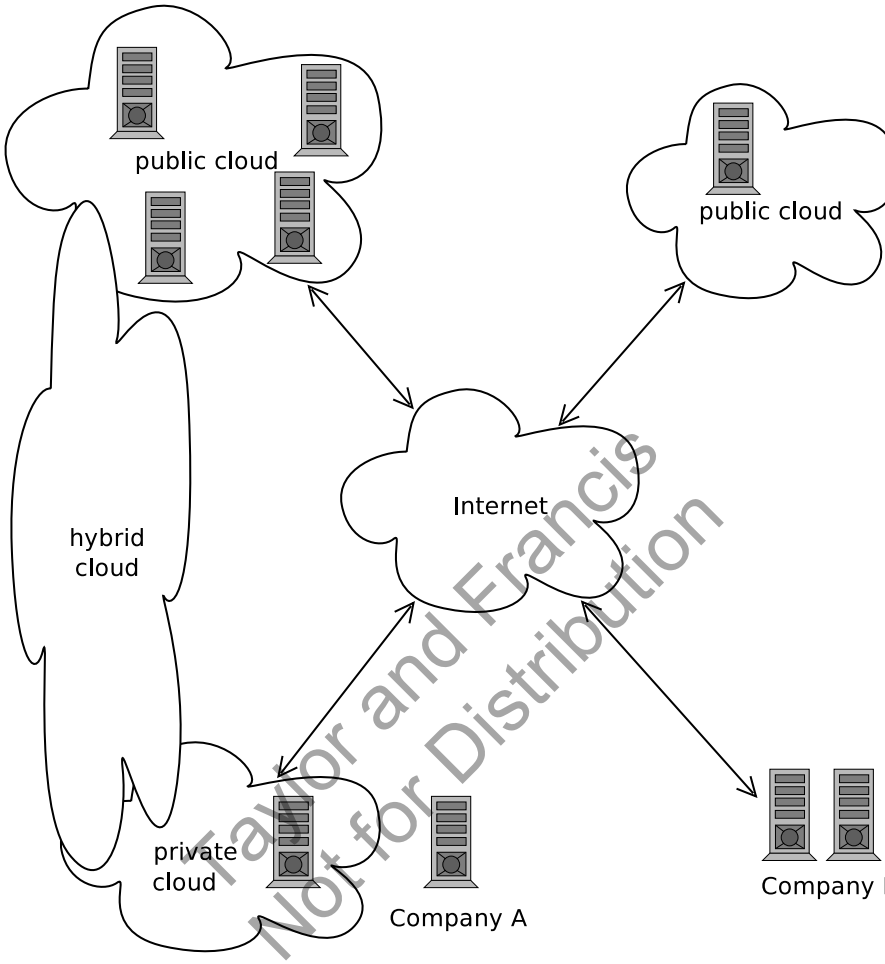

FIGURE 2.10: Cloud computing with private, public, and hybrid clouds.

2. *Risk of vendor lock-in*: If the vendor started changing conditions considerably, the client may find it difficult to migrate to another cloud provider if too invested; it might be time consuming and costly (however some providers use compatible solutions).

Based on available solutions and particular offers including parameters and prices as well as specific functions, a client may create a configuration possibly using services from many providers [189].

Alternatively, institutions my want to set up private clouds to mitigate these issues. The following types of clouds can be distinguished in this respect:

- *Public cloud*, in which case an external provider offers access to a cloud or clouds managed by them.
- *Private cloud*, which is maintained by the given institution and offered as a service within this institution. If the latter is large, it may turn out to be an efficient way of resource management for various geographically distributed subsidiaries. At the same time, the cloud is managed and controlled by the same institution. However, it is the institution that needs to purchase and maintain both hardware and software in this case.
- *Hybrid cloud*, in which particular services may be provided from such a location as to optimize factors required by the client; as an example:
  1. An application for which high security is required, may be provided from a private cloud while infrastructure for time-consuming but not that critical computations may be used from a public cloud.
  2. An enterprise can continue to store sensitive data of its current clients and contracts in a private cloud and offload less critical archive data to a public cloud.
  3. An enterprise interested in particular software typically offered by a public cloud may use a private cloud for this software due to security reasons.

The following sections present types of cloud services offered on the market. Each of the services is provided based on a contract with a certain duration and certain QoS terms, including the price.

2.1.7.1 Infrastructure as a Service

Infrastructure as a Service (IaaS) is a type of a cloud service that offers clients the requested resources in a particular configuration. The client can request particular parameters with respect to the following:

1. Number of requested nodes/computers
2. RAM size
3. Storage/disk space
4. CPU power
5. Operating system, e.g., Microsoft Windows, Linux etc.
6. Outgoing and incoming transfer per month
7. QoS parameters such as requested availability, e.g., 99.9%

What is important is that the client can usually scale up and down the requested configuration based on current needs and pay only for the configuration currently needed. Examples of particular cloud-based compute-type solutions include Amazon Elastic Compute Cloud (Amazon EC2) [156] and Google Compute Engine service [171]. Another solution is Eucalyptus (with support for Amazon EC2 and Amazon S3 interfaces) which allows creation of private and hybrid clouds [90]. OpenStack [14, 179] allows you to manage compute, storage, and networking resources and is a platform for private and public clouds. It also provides an API compatible with Amazon EC2.

2.1.7.2 Software as a Service

Software as a Service (SaaS) is a type of cloud service that offers clients access to particular software. The software is maintained by the cloud provider who performs all necessary updates while the client can use the service from any location. Most often, *flat payment* for the services in the given configuration is assumed and for the duration of the contract.

2.1.7.3 Platform as a Service

Platform as a Service (PaaS) is often referred to as exposing a complete hardware and software platform as a service to clients. For instance, a particular complete programming environment with a requested operating system may be offered on an appropriate hardware system to programmers with a team work environment. Examples include Red Hat's OpenShift [174] and Aneka [135, 170].

2.1.7.4 Cloud vs. Grid Computing

There are a few important differences when compared to grid systems [104]:

1. Business goals: While grid systems emerged as distributed systems for integration of high-performance resources of mainly institutions, cloud computing's targets are both businesses and end users who outsource hardware, software, or hardware and software platforms.

2. API: Grid systems use grid middleware that can use queuing systems to access particular computational resources. Grid middleware expose job management, file management, monitoring and security APIs as explained in Section 2.1.5. Cloud systems use specific APIs or interfaces such as Amazon EC2 and Amazon S3.

3. Organizational: In grids: cooperation including allowing access to use resources, authentication and authorization of users is agreed between Virtual Organizations. In cloud computing, providers generally attract clients independently.

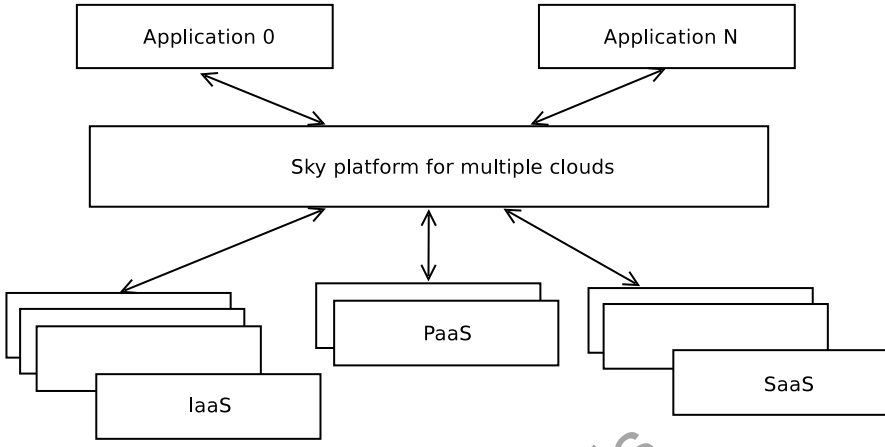

FIGURE 2.11: Sky computing on top of multiple clouds.

2.1.8 Sky Computing

Obviously, there are many IaaS, SaaS and PaaS providers. This means that, similarly to services in SOA, a client can select the best cloud providers based on required QoS parameters. For instance, Cloudorado [5] allows users to find the best IaaS offers based on the client's requirements.

Sky computing [127] can be seen as an extension and generalization of cloud computing to overcome the limitations of the latter. Namely, the client uses an infrastructure that connects to multiple clouds to use required services as shown in Figure 2.11. Thus sky computing leads to creation of a middleware on top of cloud providers. The goal of this middleware is similar to the functions of the grid middleware. Similarly, issues such as the following ones appear:

1. lack of a uniform interface to various clouds (some use the same interface as explained in Section 2.1.7) as well as security management,
2. difficult communication and synchronization between clouds; there are network restrictions for particular clouds related to configuration of VMs, IPs, high numbers of hops between machines [95],
3. difficult data management across multiple clouds at the same time.

Compared to clouds, sky computing offers the following benefits:

1. Independence from a single cloud provider: In case one cloud changed its conditions considerably, the client can simply select another cloud

provider without much effort; this is obviously possible if the APIs for the clouds are the same or similar.

2. Promotes increased competition from other cloud providers that can be used by a particular client. Sky computing mitigates the problem of *vendor lock-in* in this way. This is beneficial for the client and the provider might be well aware of other options for the client.

3. Ability to offer a set of services on top of clouds that would not be available from a single cloud provider

4. Ability to use various clouds from possibly various providers for particular parts of a complex scenario.

2.1.9 Mobile Computing

The recent years have marked fast growth of the mobile device market. The following important software areas for mobile computing could be distinguished since mobile devices became popular recently:

1. personal information management, scheduler, office applications,

2. client to WWW servers and Web Services,

3. location-based services based on the position of the mobile device,

4. sensors used in smart homes/world such as GPS, acceleration, light, temperature, etc.

The latter is also related to *ubiquitous computing* in which the distributed system is filled in with sensors and proper services are invoked based on the current context. For instance, when the owner of an apartment approaches, the door is unlocked and music is turned on based on the current mood of the owner, temperature, humidity and the time of day. Mobile devices with a wide array of sensors may be used for this purpose as well.

Naturally, mobile devices can communicate with various services around and even far away from them. Two types of services can be distinguished:

1. Those offered by other mobile devices using, e.g., Bluetooth.

2. Services offered from stationary servers, e.g., Web Services, servers (e.g., mail) allowing socket connections, POP, asynchronous notification using, e.g., Google Cloud Messaging (GCM) [114], etc. This can allow integration of mobile devices with weather forecasts, HPC systems for processing of digital media content from cameras, etc.

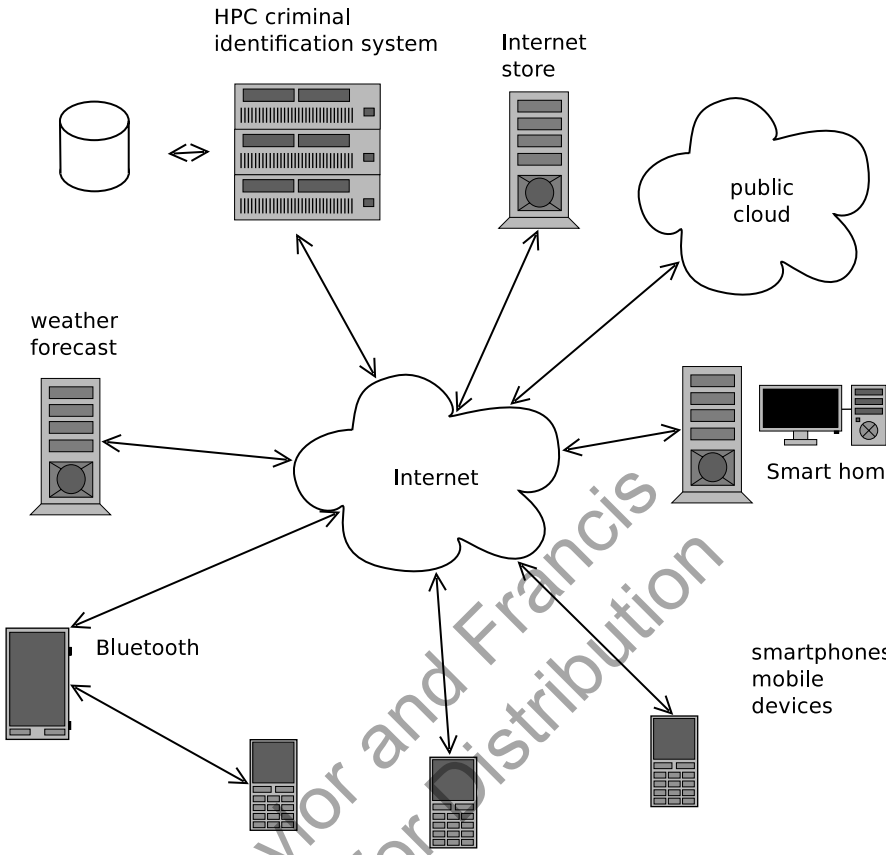

FIGURE 2.12: Mobile computing.

2.2 Complex Distributed Scenarios as Workflow Applications

Today's distributed systems are all about service integration as this leads to added value and possibilities compared to individual services. *Workflow management systems* have been introduced to model, optimize, execute and manage complex scenarios in information systems. In general, a complex scenario is represented by a *workflow application* or a *workflow* which can be modeled as a graph in which nodes correspond to tasks which are parts of a complex process while the edges denote control and corresponding data flows. This approach applies, with slight differences, to complex tasks in the following areas:

1. Business processing in which tasks can be performed by various services offered on the market by various providers on different terms. Business workflows are more about control flow rather than processing large amounts of data; various control mechanisms should be predicted including exceptions.

2. Scientific computing in which workflow tasks are usually to process large amounts data; parallel paths of the workflow can be used in order to parallelize processing; control flow is rather simple.

3. Ubiquitous computing in which the graph depicts sequences of invocations of services based on the context, i.e., a set of conditions that initiates a given service or services.

Workflow applications can be characterized in terms of many features. Typical aspects applicable to many contexts are discussed in subsequent sections.

For applications run on certain system types, thorough taxonomies have been proposed, e.g., for grid workflow applications in [216, 222].

2.2.1 Workflow Structure

Depending on the particular field of study, various constructs are allowed in workflows which determine possible control and data flow as well as optimization and execution. Typical constructs may include:

1. *sequence (S)*—tasks connected with a directed edge are executed in the specified order,

2. *choice (C)*—one of alternative parallel paths can be executed,

3. *parallel execution (P)*—two or more parallel paths may be executed at once,

4. *iteration (I)*—a part of a workflow may be executed many times.

The most popular formulation of a workflow is a directed acyclic graph (DAG) which includes constructs S, C and P.

2.2.2 Abstract vs. Concrete Workflows

Given a particular workflow structure, there are two ways that executable components can be assigned to particular workflow nodes. This determines the type of workflow:

- *Concrete*: In this case it is specified a priori what executable code is assigned to particular workflow tasks. This might be given in various forms:

1. for each workflow node an application (executable) is assigned that executes on a particular resource (computer),
2. an external concrete service is assigned to a workflow node.

- *Abstract*: In this case the workflow definition for each workflow node contains specification of what task the node should do. This can include:
  1. a *functional* description of what the particular workflow node should perform; one of many possible executables or services can be selected to accomplish this task,
  2. *non-functional (QoS)* requirements imposed on the workflow task, e.g., that it must finish within a specified time frame or must not be more expensive than a given threshold.

2.2.3 Data Management

Data management is very important for workflow applications oriented on both:

1. data flow, where options for partitioning of large amounts of data for parallel processing are crucial,
2. control flow, where, e.g., making a choice based on input data is made.

The literature discusses several constructs and approaches: various ways of data integration coming from various inputs [103], data representation such as files, or data streams [60, 106].

It should be noted that workflow tasks may be assigned particular data and correspondingly data sizes in advance. In such cases the problem is usually to determine which service should perform the given task such that a function of quality metrics is optimized.

Many other problems could be considered as well. For instance, if input data of a given size needs to be partitioned among workflow paths for parallel execution in such a way that capacity constraints need to be met and flow conservation equations need to hold as well as the cost of processing is to be minimized, we would consider the minimum cost flow problem [203].

2.2.4 Workflow Modeling for Scientific and Business Computing

The most popular model of a workflow application used in scientific and business applications is an abstract DAG workflow. This allows flexibility in the definition of a complex task, allows selection of appropriate services, and stating a practical optimization problem for subsequent execution. The process involves a few, steps:

1. Workflow definition
2. Finding services capable of executing workflow tasks
3. Selection of services for particular workflow tasks and scheduling service execution at particular moments in time
4. Execution
5. Monitoring the status and fetching output results

Formally, an abstract DAG-based workflow considered in the literature is represented as a directed acyclic graph $G(T, E)$ where T is a set of tasks (graph nodes) and E a set of directed edges connecting selected pairs of tasks. For each task t_i there is a set S_i of services s_{ij}s each of which is capable of executing task t_i. The notation corresponding to this model is as follows:

- t_i—task i (represented by a node in the workflow graph) of the workflow application $1 \leq i \leq |T|$
- $S_i = \{s_{i1} \ ... \ s_{i|S_i|}\}$—a set of alternative services each of which can perform functions required by task t_i; only *one* service must be selected to execute task t_i $1 \leq i \leq |T|$
- $c_{ij} \in R$—the cost of processing a unit of data by service s_{ij} $1 \leq i \leq |T|$ $1 \leq j \leq |S_i|$
- $d_i = d_i^{\text{in}} \in R$—size of data processed by task t_i $1 \leq i \leq |T|$
- $t_{ij}^{\text{exec}} = f_{ij}^{\text{exec}}(d_i^{\text{in}})$—execution time of service s_{ij}
- $t_i^{\text{st}} \in R$—starting time for service s_{ij} selected to execute t_i $1 \leq i \leq |T|$
- $t_{\text{workflow}} \in R$—wall time for the workflow i.e., the time when the last service finishes processing the last data
- $B \in R$—budget available for the execution of a workflow

Examples of practical workflows for three application areas are as follows:

- *Business*—a workflow application that can be regarded as a template for development and distribution of products out of components available on the market; generalization of a use case considered in [66] and extended with dependencies between services (Figure 2.13):
  1. The first stage distinguishes parallel purchases of components from the market; for each one there may be several sale services available with different QoS terms such as delivery time and cost.
  2. The product is integrated out of the collected components with the know-how of the workflow owner (company).

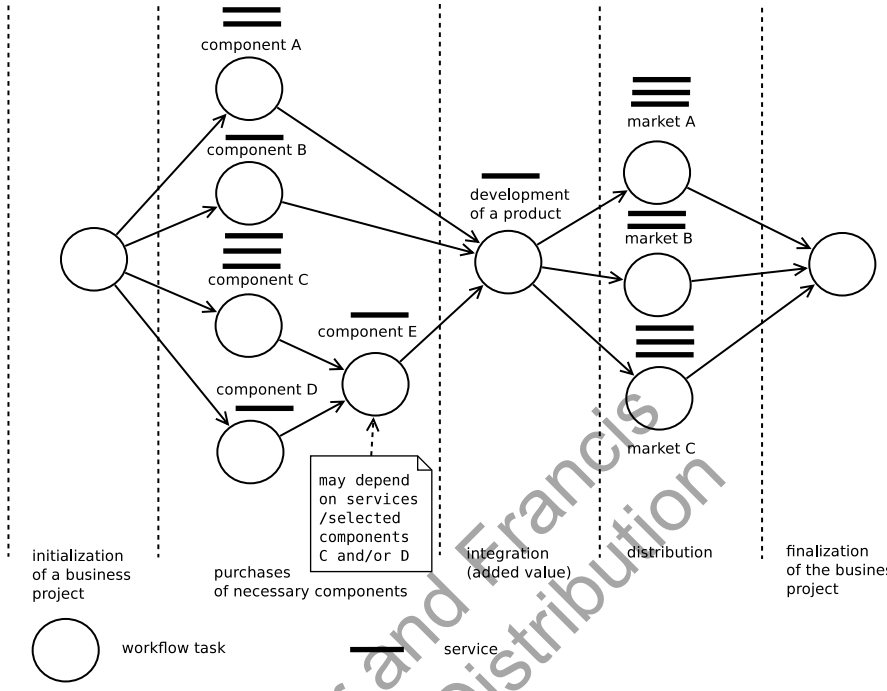

FIGURE 2.13: Business workflow application example.

3. The product is distributed on several, potentially geographically distributed markets; for each market there may be several distributors the product developer can choose from.

- *Scientific*—a workflow for parallel processing of input data shown in Figure 2.14 that can be used for a variety of applications such as parallel processing of aerial images, signals from space, recognition of illnesses based on patient's images and a database of templates, voice recognition, detection of unwanted events [74]. The author distinguished a generalized template with several stages that fit and can be adjusted to all these applications:

  1. Data acquisition—input data can be gathered in parallel from various sources such as cameras, audio input devices, etc. it can be noted that there can be both concrete tasks for which there already known data acquisition services (such as a particular camera) or abstract tasks; for the latter, there can be various services that provide services, e.g., services from various market analysis providers.

2. Data set preparation—acquired data is arranged into data sets for parallel processing; note that both various data sets can be processed in parallel as well as each individual data set may be parallelized in the next stage.
3. Parallel processing of particular data sets. It should be noted that in this workflow formulation it is possible to process various data sets in parallel; in particular, acquisition of another data set (such as from source C) does not synchronize with processing of data from source A; parallel processing has been arranged into parallel workflow paths which are realized by particular services installed on supposedly various resources.

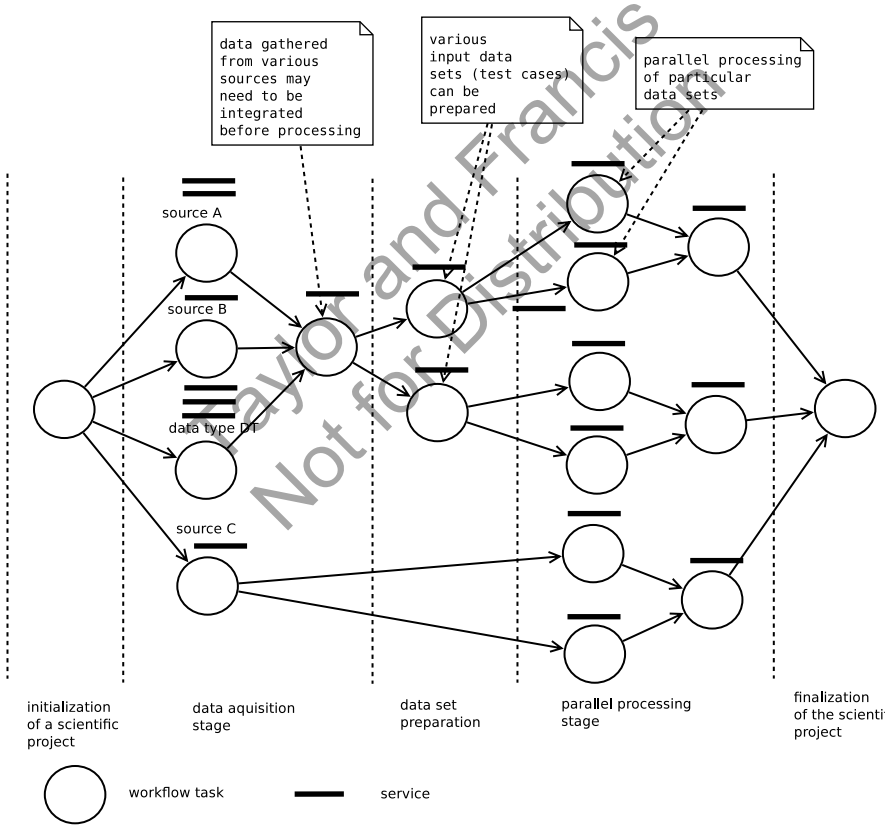

FIGURE 2.14: Scientific workflow application example.

2.2.5 Workflow Scheduling

For abstract workflows that contain tasks with two or more services assigned to them, a workflow scheduling problem needs to be solved. Given the aforementioned dependency constraints defined by the graph, using the introduced notation, the workflow scheduling problem can be stated as follows:

$$\forall_i \text{ find } t_i \rightarrow (s_{i\ sel(i)}, t_i^{\text{st}}) \tag{2.1}$$

where $s_{i\ sel(i)}$ denotes the service selected for execution of t_i starting at time t_i^{st} such that

$$\forall_{i,k:(t_i,t_k)\in E}\ t_k^{\text{st}} \geq t_i^{\text{st}} + t_{i\ sel(i)}^{\text{exec}} \tag{2.2}$$

with constraints on the workflow execution time and cost. The workflow execution time (not considering scheduling time) is the end time of the latest task with no successor:

$$t_{\text{workflow}} = max_{i:\nexists k(t_i,t_k)\in E}\{t_i^{\text{st}} + t_{i\ sel(i)}^{\text{exec}}\}. \tag{2.3}$$

Typically, considered alternative optimization goals include one of the following:

- *MIN_T_C_BOUND—minimization of the workflow execution time with a bound B (budget) on the total cost of selected services:*

$$\begin{aligned} &min\ t_{\text{workflow}} \\ &\sum_i c_{i\ sel(i)} d_i \leq B. \end{aligned} \tag{2.4}$$

- *MIN_C_T_BOUND—minimization of the total cost spent with an upper bound on the workflow execution time T:*

$$\begin{aligned} &min\ \sum_i c_{i\ sel(i)} d_i \\ &t_{\text{workflow}} \leq T. \end{aligned} \tag{2.5}$$

- *MIN_H_C_T—minimization of a function of cost and execution time (assuming certain units)*:

$$min\ h(\sum_i c_{i\ sel(i)} d_i, t_{\text{workflow}}). \tag{2.6}$$

Formulation 2.6 might be useful if we consider a financial equivalent of time which is a very practical approach applicable to, e.g., transportation [152]. Formulations such as 2.4 and 2.5 are NP-hard problems [225] which means that

optimal solutions can be found in a reasonable time frame only for problems of small size. For larger problems that show up in real-life situations, efficient heuristic algorithms need to be adopted. Selected algorithms for solving these problems are discussed in Section 4.2.

2.2.6 Static vs. Dynamic Scheduling

It should be noted that although the workflow scheduling problem considers a set of services for optimization, availability of these services might change. In general, in real environments, the following types of events might occur and impact scheduling results:

1. Some services that were previously chosen for execution are no longer available. This might be due to failure of a network, the server on which the service is installed or possibly even maintenance.
2. Changes in service parameters unless a contract was put in place that would guarantee expected values for QoS metrics such as execution time, price, etc.
3. New services showing up on the market that would be capable of executing workflow tasks on better terms.

In cases 1 or 2, rescheduling of the workflow has to be performed in order to meet the optimization goal with possibly alternative services. In case 3, the cost of rescheduling should be assessed as compared to potential gains from using services with potentially better parameters.

It should be noted that there can be many more parameters/quality metrics assigned to services, apart from cost c_{ij} and time t_{ij} such as dependability, reliability, security, being up-to-date, etc. These are discussed in Section 2.3.3. All of these can also be incorporated into either the aforementioned or dedicated constraints.

2.2.7 Workflow Management Systems

There are several workflow management systems available which can be characterized in particular by [222]:

- *Target application type*—business, scientific, ubiquitous computing
- *Underlying type of services*—e.g., one of the following: Web Services, grid or cloud services
- *Manual composition of tasks into workflows or automatic based on existing rules and facts*
- *Optimization* of a single criterion or multiple criteria at the same time

- *Scheduling*—using local or global knowledge
- *Various ways of monitoring and learning* about services and providers, etc.

Exemplary workflow management systems that can be distinguished based on the type of target systems include:

- *Grid*—usually built for grid systems on top of grid middleware such as Globus Toolkit, Gridbus, etc. or services. Examples of such systems include ASKALON [215], Taverna [18], Pegasus [83, 84], Triana [20, 150], Kepler [11, 146], Conveyor [142], and METEOR-S [137], which allows adding semantics to web service composition with quality of service [27, 176].
- *Cloud*—recently, cloud systems have become attractive platforms for both business and scientific services [45, 63] with emphasis on the payment-per-use policy for cloud systems compared to grid systems [116, 211] and possibility of launching various configurations easily. Another related advantage is scaling resources as needed, which can be useful for scientific workflow applications with a varying level of parallelism such as different numbers of parallel paths in various stages. Dynamic scaling is possible in the Aneka Cloud [170]. Cluster-based systems offer better communication performance at the cost of flexibility [123]. Examples of systems include:
  - Amazon Simple Workflow [40]—allows definition and management of business workflows on clouds and/or traditional on-site systems.
  - Tavaxy [26]—a system that integrates Taverna and Galaxy for launching a part or a whole workflow in a cloud.
  - A framework and workflow engine for execution of workflows on many clouds [101].
- *Ubiquitous computing*—services are invoked in the given context. FollowMe [140] is an example of a platform for this type of computing.

Similarly, various languages and standards for representation of workflow applications are used in various contexts [104], for example:

- *Grid*—Petri-Nets (e.g., Triana) [150], Abstract Grid Workflow Language (AGWL) [91] used in ASKALON
- *Business*—Business Process Modelling Notation (BPMN) [165] with XML Process Definition Language (XPDL), Web Services Business Process Execution Language (WS-BPEL) [205], OWL-WS [41] (NextGrid). Orchestra [16] is a solution for complex business processes with WS-BPEL 2.0 support. BPELPower [221] was designed and implemented in

Java as a solution that can run both typical BPEL and geospatial workflows with handling Geography Markup Language (GML) and geospatial Web services. ApacheODE [2] is able to execute processes expressed using WS-BPEL 2.0 and the legacy BPEL4WS 1.1. Activiti [1] is a platform with a BPMN 2 process engine for Java. jBPM [10, 148] is a Business Process Management (BPM) Suite with a Java workflow engine for execution using BPMN 2.0. Execution using BPEL is considered in various contexts such as grids [147] or mobile devices [109].

- *Ubiquitous computing*—Compact Process Definition Language (CPDL) in FollowMe [140].

SHIWA (SHaring Interoperable Workflows for large-scale scientific simulations on Available DCIs) [24] allows you to store various workflows in a repository and run these on various Distributed Computing Infrastructures (DCIs). What is novel is that the approach enables you to define higher-level meta-workflows that consist of workflows run on various workflow management systems. This greatly increases interoperability among workflow-based solutions. Furthermore, Askalon was extended for use in grid-cloud environments [167] with an Amazon EC2-compliant interface. Launching a workflow on various systems including clouds, grids and cluster-type systems is shown in [170] with support for Aneka, PBS, and Globus.

2.3 Challenges and Proposed Solutions

2.3.1 Integration of Systems Implementing Various Software Architectures

As described in Section 2.1, various software architectures of distributed systems were designed for particular types of processing and applications. Consequently, integration of various components within a single software architecture and its actual implementation is straightforward. For example:

1. HPC: MPI allows parallel processing and synchronization of processes running on various nodes or processors.
2. HPC: NVIDIA CUDA and OpenCL allow parallel processing using multiple threads running on many Streaming Multiprocessors (SM). The same technologies allow parallel usage of multiple GPUs at the same time.
3. SOA: By design, a Web Service can consume and integrate results of other Web Services. Web Services are loosely coupled and can be offered by various providers, from any place. Cooperation is possible using the well-established SOAP-based or RESTful services.

4. Multitier Architectures: Within Java Enterprise Edition [120], components of the presentation layer such as servlets and JSPs cooperate easily with Enterprise Java Beans in the business layer. Furthermore, components can interact across application servers thanks to clustering within this technology [169].

5. Grid systems: By design, systems and software of distributed Virtual Organizations (VOs) is coupled together and made available using a uniform API and interface that is called *grid middleware*. As an example, Globus Toolkit provides a uniform API for management of jobs (tasks) on the grid such as clusters located in various VOs through GRAM, file management through GridFTP, and security using GSI.

6. Cloud computing: Various resources offered by a cloud provider can be used together seamlessly and in a way not visible to the client, e.g., in Google Apps service.

7. Mobile computing: Well-established standards such as Bluetooth, and HTTP serve as common communication protocols for various mobile devices. Nevertheless, even though Bluetooth is a well-defined standard, communication glitches including disconnecting, and problems when establishing communication are common when using devices such as mobile phones and GPS navigation devices from various manufacturers. Other standards such as Web Services, and communication over sockets can be used as well-provided there is an implementation for the devices in question.

Problem: Integration of applications or components running within software systems implementing *different* software architectures is not always possible out-of-the-box. There do exist some approaches in various contexts. For instance, *sky computing* [127] aims at integration of various clouds from various providers. In an enterprise environment, Enterprise Service Bus (ESB) allows integration and communication between various applications and services using various communication protocols and formats [182, 202]. Various topologies are possible for integration of, e.g., separate subsidiaries: a unified or separate ESBs [143]. A BPEL engine that integrates services can be either independent from an ESB and can call services from various ESBs or operate within an ESB [143]. As [182] suggests, extension of an ESB implementation with new protocols is not easy as it demands a new port type. A concept of the universal port is proposed along with protocol and format detectors and processors for use with an ESB. Furthermore, it is always possible to use certain technologies such as Web Services in order to create a middleware hiding differences between certain types of systems standing behind it.

However, at the level of the aforementioned software systems implementing particular software architectures there is no solution that would address all aspects of uniform integration. Such a solution requires integration at the level of protocols, and data formats, but also authentication, authorization,

potential queuing systems and potentially complex APIs exposed by those software systems.

Rationale: For many modern applications, integration of various distributed systems is desirable. For instance:

1. business applications using services from various companies, entities including government and local authorities,
2. scientific computing for gathering data from, e.g., distant radio telescopes from all over the world and processing in parallel on various clusters,
3. ubiquitous computing when local services are discovered and used dynamically in a workflow application based on the physical location of the device.

As an example, the scientific workflow shown in Figure 2.14 may use services for parallel processing of data, whether on an HPC system with the PBS queuing system, available as a Web Service from a distant server, available through Globus Toolkit, or processed in parallel by many volunteers attached to BOINC. From the perspective of the workflow modeling, it does not affect the structure or definition of the workflow in any way. Such mapping of software implemented using various architectures could prove useful.

Proposed solution: The author proposes to use a generalized and uniform concept of a *service* for *any* software components implementing these software architectures to be able to integrate actual resulting services into complex scenarios modeled by workflow applications. In fact, the concept of a service is already present in, e.g.,:

1. SOA—as a basic software component and implementation through the Web Service [88]
2. Grid computing—the concept of a *grid service*
3. Cloud computing—the concept of SaaS

Some other architectures and actual implementations require publishing of selected legacy software components as services. The actual solution is presented in Sections 3.1, 3.2, and 3.3.

2.3.2 Integration of Services for Various Target Application Types

Essentially, software services can be divided into categories in terms of target application types:

1. Business—when the functional goal of the service is customer oriented with QoS metrics such as performance, cost, reliability, conformance, etc. are very important for the client.

2. Scientific— when the primary goal of the service is to perform computations, data analysis, predictions, etc., that do not need to be directly applicable or purchased by an average consumer. Usually such services are performance oriented although the cost and power consumption are gaining attention as shown in Section 2.1.1.

3. Ubiquitous— services oriented on consumers but usually responding to frequent requests with fast replies and responding in the given *context*.

Problem: Provide a concept of a software component that would contain a uniform description of the purpose, target environment, and handling a request and response as well as offer a uniform API for client systems.

Rationale: Many modern interdisciplinary problems require services from various fields of study. For instance, design of a modern aircraft would require simulations related to strength, durability, reliability of various components that would need to be performed on HPC resources, most likely in different, geographically distributed centers, purchase and service cost analysis, marketing services and many others. This will certainly involve business, and scientific services performed both by software and human specialists.

Proposed solution: The author proposes a uniform description of the *service* as a component suitable for all these target uses. The description would contain all details relevant for all these target applications. The actual solution is presented in Sections 3.1, 3.2, and 3.3.

2.3.3 Dynamic QoS Monitoring and Evaluation of Distributed Software Services

Uniform description of services targeting various applications and internally implemented using various software architectures does not make these practical to use until one can reliably assess their qualities. What is more, it can be seen clearly in today's distributed software market that static assessment of the service QoS is not enough. Table 2.2 lists QoS metrics for which precise, reliable, and up-to-date assessment for particular service types is crucial. Thus, the following can be stated:

TABLE 2.2: Important QoS metrics for particular types of services.

Service Type	QoS Metric	Notes
HPC	Performance	Usually HPC is used to shorten the execution time
	Dependability	One expects to trust the returned results
	cost	Usually needs to be below the predefined threshold

	Availability and reliability of a service	What the customer expects from the service
SOA	security	All communication between the client and the service should be encrypted and data processed and/or stored in a secure way
Multi-tier Systems	Ease of use	Clear and friendly API
	Availability	From the customer's point of view
Grid	Ease of use	Availability is supported by the grid middleware that would make use of available services on attached Virtual Organizations
	Security	Crucial when cooperating with other organizations
Volunteer	Performance	The main reason to search for others' computing resources
	Reliability	Results need verification against hardware/software errors and potential harmful users' intentions
Cloud Computing	Security	Crucial for the client because the data is managed by a third party
	Availability of alternatives	To avoid vendor lock-in
	cost	The client wants to minimize it
	reliability	The customer expects the service (IaaS, SaaS, PaaS) to be working at any time within Service Level Agreement (SLA)
	Performance	The client wants to maximize it within the cost constraint
Mobile Systems	Availability	The customer expects the service to be mostly accessible
	Cost	May be a decisive factor for choosing the service as there may be many alternatives
	Location awareness	May help optimize the result of the service
	being up-to-date	Must be based on up-to-date data, e.g., latest maps, information about Points of Interest (POI), etc.

Problem 1: Perform periodic, reliable monitoring and analysis of QoS metrics important for particular types of services. If needed, prepare forecasts for QoS values in the future that result from the past values.

Problem 2: Perform periodic, up-to-date ranking of services (possibly implemented on various architectures) that perform the given function considering past history of quality evaluation.

Rationale: Most of the aforementioned QoS metrics for the distinguished types of services change in time. Thus, it is necessary to:

1. discover new services dynamically,
2. monitor QoS of services dynamically, which may hint that some previously preferred services are no longer available.

Furthermore, it is necessary to evaluate services that are capable of performing the function requested by the user. This leads to ranking services. Depending on particular needs, various algorithms may be necessary also considering the history of QoS evaluations for the given service.

Moreover, from a global point of view, the ranking scheme may need to assure that no single provider dominates the market with their services to avoid the vendor lock-in problem in cloud computing.

Proposed solution: The suggested uniform definition of the service is extended with the procedure that describes how:

1. particular QoS metrics of the service will be measured, including dynamic measurements and application of digital filters;
2. aggregation of QoS measurements into a ranking of services in the given category.

The actual solution is presented in Section 4.1.

2.3.4 Dynamic Data Management with Storage Constraints in a Distributed System

Today, data handling becomes one of the key concerns. As computing power is becoming available almost everywhere (clusters, servers, personal computers, multicore tablets and smartphones), data management, storage constraints and moving data toward compute devices to optimize QoS criteria, especially in complex workflow applications becomes crucial. This holds true not only for particular types of systems but especially for integration of the aforementioned types of systems including sky computing.

Problem 1: Consider storage constraints in execution of complex scenarios that incorporate distributed services. Consider data communication costs in optimization of execution of the scenarios, especially location of management units and data caching.

Problem 2: Consider different ways of handling data: *synchronized* or *streaming* in various stages of a complex distributed scenario.

Rationale: Although larger and larger storage spaces are available and the price per storage is reasonably low, it is not free. It may be especially important for scenarios which handle large amounts of data, e.g., processing sales, stock data in data centers, and multimedia data out of cameras installed in malls, city centers, and along roads. Especially the latter would need transfers of large amounts of data from distributed sources, staging to computational resources, caching, parallel processing, and storage of data from a certain period from the past. Big data processing is one of the directions on which the current research in information systems is focused [153, 33].

Proposed solution: An integrated solution that distinguishes:

1. storage constraints of the resources on which services are installed,
2. storage constraints of the system that executes various services of a complex scenario and transfers data from one resource to another, and
3. caching of intermediate data transferred by the workflow management system in a dedicated cache storage.

The actual solution is presented in Chapter 5.

2.3.5 Dynamic Optimization of Service-Based Workflow Applications with Data Management in Distributed Heterogeneous Environments

Problem: Define a quality model, a dynamic optimization problem and solutions for complex scenarios that would correspond to real-life, often interdisciplinary, complex tasks that couple services from various domains.

Rationale: Complex interdisciplinary tasks often require services from various fields of study. These services differ in terms of the

1. application domain,
2. implementation platform,
3. important QoS metrics, and
4. evaluation criteria of particular services.

Proposed solution: A workflow scheduling model that integrates all of the following:

1. a model that considers a uniform concept of a service as indicated in Section 2.3.1 with a description independent from the application target as indicated in Section 2.3.2,

2. runtime monitoring and evaluation of services as suggested in Section 2.3.3,

3. definition of the workflow scheduling problem considering dynamic evaluation of services and rescheduling along with management of data processed by services as hinted in Section 2.3.4.

The detailed solution is presented in Section 3.4 with algorithms presented in Chapter 4.