



IP-FP6-015964

AEOLUS

Algorithmic Principles for Building Efficient Overlay Computers

Deliverable D6.1.2

Overlay Computing Platform: Design Report

Responsible Partner: University of Patras (EL)
Report Preparation Date: January 2007 (revised version)

Contract Start Date: 01/09/05 Duration: 48 months
Project Co-ordinator: University of Patras

Contents

1	Vision and basic design goals	1
1.1	Some specific services and related functionalities	1
1.2	A preliminary architectural view	3
1.3	Design report overview	4
2	The Overlay Computing Platform	5
2.1	The underlying platform	5
2.2	OCP architecture	6
2.3	OCP functionalities	7
2.3.1	The core layer	8
2.3.2	The service layer	11
3	Functionalities for data/resource management	13
3.1	Store service	13
3.2	Replication service	14
3.3	Indexing service	14
3.4	Naming service	16
3.5	Distributed catalogue service	16
3.6	Query service	16
3.7	Groups and resource addressing	17
3.8	Summary	18
4	Functionalities for communication and distributed computations	19
4.1	Basic communication primitives	19
4.2	Schedulers for application servers	20
4.3	Performance-conscious allocation of distributed computations	21
4.3.1	Discovery of performance characteristics	22
4.3.2	Using local non-busy cycles to gain performance information	23
4.4	Implementation of distributed JVM for thread migration	24
4.5	Summary	24
5	Functionalities for security and trust management	26
5.1	Anonymous communication network	26
5.2	Certified information access	26
5.3	Reputation management for content filtering	28
5.4	Reputation management for resource allocation	30
5.5	Secure broadcast (Byzantine agreement)	31
5.6	Secure function evaluation with limited trusted third party	32
5.7	Secure function evaluation	33
5.8	Timestamping	34
5.9	Summary	37

6	Functionalities for wireless subnetworks	38
6.1	Buffering services	40
6.2	Gateway for environmental monitoring & actuator control	40
6.3	SQL-like functionality	41
6.4	Network statistics & management / control	42
6.5	Virtual sensor networks	42
6.6	High-level network programming	43
6.7	Routing	43
6.8	Mobility prediction & control	43
6.9	Support management	44
6.10	Energy management / topology control	45
6.11	Clustering / grouping	45
6.12	Time synchronization	46
6.13	Localization	46
6.14	Code update	47
6.15	Data propagation & query dissemination	47
6.16	Sensing / monitoring & actuator control	48
6.17	Summary	49
A	JXTA at a glance	50

The recent explosive growth of the Internet gives rise to the possibility of a global computer of grand-scale consisting of Internet-connected computing entities (possibly mobile, with varying computational capabilities, connected among them with different communication media), globally available and able to provide to its users a rich menu of high-level integrated services that make use of its aggregated computational power, storage space, and information resources. Achieving this efficiently and transparently is a major challenge that can be overcome by introducing an intermediate layer, the overlay computer.

The goal of AEOLUS is to investigate the principles and develop the algorithmic methods for building such an overlay computer that enables this efficient and transparent access to the resources of an Internet-based global computer. One of the concrete objectives is to implement a set of functionalities and integrate them under a common software platform in order to provide the basic primitives of an overlay computer, henceforth called Overlay Computing Platform. This document is the design report of the Overlay Computing Platform.

1 Vision and basic design goals

We view the Overlay Computing Platform (OCP in the following) as an extension of the software environments typically used when we are working with our PCs. In the latter case, we have access to a disk space, some computational power, access to applications running on our machine, access to the Internet, etc. OCP will give the user the impression of accessing a multiuser system with a huge disk space and enormous computational power.

From the programmer's point of view, OCP looks like a very powerful computer with many (probably heterogeneous) nodes with the ability of running parallel applications transparently. Transparently means that the programmer can only define the desired properties of the computers he wishes to run his processes; the OCP should monitor the status (i.e., load, available space, bandwidth of incoming/outgoing links, etc.) of several nodes, find the ones with the desired properties and transfer the computation on these nodes. Furthermore, when required by the applications (e.g., by a global data sharing and information query application), the OCP will be able to adapt its configuration. The OCP will also support primitives for secure communication.

1.1 Some specific services and related functionalities

We see at a first level of the OCP some primitive functionalities which support the ability of the OCP to create groups of computers (running the platform), search for/discover nodes, detect failures, handle joins and leaves of nodes, monitoring status/performance, and also very basic cryptographic functionalities that are needed to get some level of security.

In a second level we see functions handling storage and computing resources, such as the following two scenarios indicate:

A distributed storage scenario: One possible set of functionalities at different levels comes from a virtual organization of the aggregate OCP space as a flat space. Consider the typical structure of our data into directories, files, blocks, etc. Such an organization could be extended to more than one disk by having a dedicated space on each computer running the OCP, and a set of pointers to blocks of information into several different disks. Instead of the block identifiers that could be used in the organization of typical single-machine file systems, we also need some computer identifier, IP address, and disk identifier (in case of computers with many disks). All these should only be known to the OCP; neither the user nor the programmer needs to have any idea about them (in a similar way we do not know today at which block of the disk our files are physically stored). The OCP uses this information to organize files and directories and spread it into several available disks without revealing this information to the user/programmer. At a very low level, the OCP provides functionalities of accessing blocks in different disks of different machines (e.g., write/read a block A into location B of disk D at machine E), data replication for fault tolerance or dynamic changes, data consistency maintenance, etc. At a higher level, the OCP provides typical functionalities to access (i.e., open, read, write, close) a file just like in typical operating systems and programming languages.

A parallel computing scenario: An option the OCP should provide to the programmer is the ability to develop parallel/distributed code that is executed in many computers running the OCP. This means that the developed program consists of a “front end process” that is running on the computer of the user, and of several processes distributed to computers running the OCP. This is done transparently in the sense that the programmer may only define properties of the computers he wishes to run these processes while the user does not even have to know whether these processes are running locally or not. The OCP has to find “free” and “not-heavily-loaded” computers where the processes will be transferred and executed. Remote execution of a process can be satisfied with machine-independence by assuming that all processes are Java classes executed on the Java Virtual Machine that runs in parallel to the OCPs in each computer. For the communication between processes, a message-passing mechanism working as follows could be used. Each process has an identifier and a process can execute a `sendmsg` command in order to send a message to another process (without knowing where the other process is running). The message is actually sent to the local OCP which is responsible for forwarding the message to the correct computer whose local OCP is responsible to deliver the message to the correct process. So, the idea is that the information about which process is running at which computer is handled by the OCPs. Typical functionalities that should be present in order to support this scenario are low level functionalities for launching processes remotely (remote “fork”) and interprocess communication (send/receive message), and higher-level functionalities to support load balancing, process migration, etc., as well as monitoring functionalities.

Additional specific functionalities that have not been discussed in the above two scenarios should be included in the OCP (e.g., related to trust and reputation management, negotiation mechanisms, resource allocation, content-based routing, economic strategies, etc.), since they will be useful in many typical distributed applications .

1.2 A preliminary architectural view

As a first approximation of the OCP, in order to support scenarios like those discussed in the previous paragraph, we see at least two levels of functionalities

- functionalities executed by the OCP in order to implement primitive operations such as read/write disk blocks, message passing, remote fork, node search, join/leave, failure detection, etc.
- functionalities to which the programmer has access.

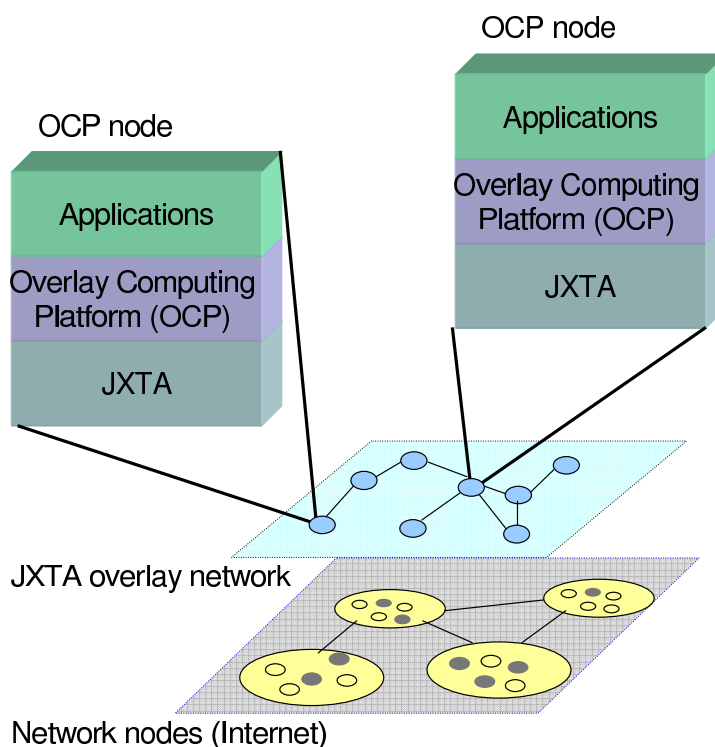


Figure 1: A preliminary architectural view of the Overlay Computing Platform.

The OCP is a distributed system. Each participating computer runs locally an OCP program to handle the issues above (and many more to be defined later in this document). We may think of this locally executed OCP program as the gateway of applications to the OCP. The OCP will use JXTA as the underlying platform; a preliminary architectural view is depicted in Figure 1.

1.3 Design report overview

Although very preliminary and informal, the discussion above reveals many of the characteristics of the OCP. The remaining sections provide a detailed specification of its architecture and functionalities. In particular, Section 2 is devoted to the architecture of the OCP, the description of its main components and an overview of the functionalities provided. Then, specific functionalities of each component are described in Sections 3–6. In particular, data/resource management functionalities are presented in Section 3, communication and distributed computation functionalities are presented in Section 4, security and trust management functionalities are presented in Section 5 and, finally, functionalities for wireless subnetworks are presented in Section 6.

2 The Overlay Computing Platform

2.1 The underlying platform

The OCP will use JXTA [1] as the underlying platform and will be built on top of it by enhancing and extending its functionalities. JXTA intends to provide a uniform programming platform for peer-to-peer applications and facilitates interoperability. It provides well structured APIs and a clear separation of concerns in each architecture but does not describe any structural or functional system properties. In particular, JXTA can support the low-level functionalities since it provides primitive functionalities such as searching for nodes, composing groups of nodes, joining/leaving groups, detecting failures, publishing services and resource description, etc. It does not explicitly provide the functionalities required for the storage and computing scenarios or functionalities for security or for the management of wireless subnetworks; these functionalities will be developed within AEOLUS. Furthermore, we have chosen the JXTA technology because it provides a set of open protocols that allow any connected device on the Internet, ranging from cell phones and wireless PDAs to PCs and servers, to communicate and collaborate in a peer-to-peer manner. JXTA is the industry-leading peer-to-peer technology, supported by over 30,000 members worldwide with more than 12 million downloads. A short overview of JXTA is presented in appendix. Note that the current stable JXTA is available for Java (J2SE, J2ME) and C/C++. In order to avoid integration problems, the main programming language that will be used in our implementation is Java.

In the selection of the underlying platform, other alternatives such as GNUUnet, DHT-based implementations, JADE, etc. have been considered. GNUUnet is a framework for secure peer-to-peer networking that does not use any centralized or otherwise trusted services. The first release of GNUUnet was published in late 2001 with a set of new technical ideas for secure peer-to-peer networking. GNUUnet uses a simple, excess-based economic model to allocate resources. Peers in GNUUnet monitor each other's behavior with respect to resource usage; peers that contribute to the network are rewarded with better service. The two main reasons that have led us not to adopt GNUUnet as the underlying platform are that: i) as part of the GNU project, GNUUnet code is continuously under development and ii) GNUUnet is not an industry-leading peer-to-peer technology. JADE (Java Agent DEvelopment Framework) is a software framework fully implemented in Java. It simplifies the implementation of multi-agent systems through a middleware that complies with the FIPA specifications and through a set of graphical tools that supports the debugging and deployment phases. JADE could support storage and computing scenarios, but it is strongly oriented towards the development of a multi-agent based system. This seems to be a new and promising programming paradigm, but, again, not widely used. DHT-based implementations will be considered since DHT-style addressing will be a service of the platform. So, existing DHT implementations can be reused here.

2.2 OCP architecture

The OCP that runs locally on each machine (OCP node) has the layered architecture depicted in Figure 2. At the bottom level, we have the network infrastructure using standard Internet protocols as well as protocols that provide access to wireless subnetworks. JXTA sits on top of this infrastructure and below OCP services. The set of functionalities provided by JXTA (some of them will be enhanced within the project) and OCP services that will be developed constitute the main “proof-of-concept” of AEOLUS. Distributed applications will run on top of the OCP.

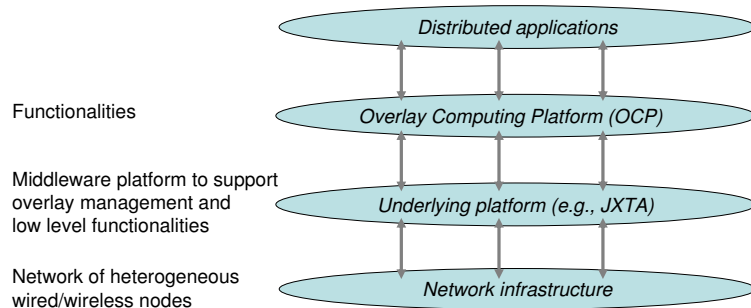


Figure 2: The system architecture

We have performed two levels of further refinement of the architecture of OCP nodes. In a first flat refinement (see Figure 3), the core OCP functionalities are integrated in the JXTA platform adding new primitives or improving the existing ones. The high level primitives are build over the JXTA platform and provide the API to be used in distributed applications.

In the structured approach (see Figure 4) the core OCP functionalities are implemented using JXTA primitives, and are well separated from the JXTA layer. The flat approach has obvious advantages in terms of performance since the core functionalities are implemented at the peer-to-peer overlay management layer. The drawback of this architecture is that the OCP will heavily depend on the JXTA technology and on a specific JXTA version. On the contrary, although the structured approach reduces system performance, there is a greater level of independence from the peer-to-peer middleware platform and version. Indeed, only the OCP core needs to be modified in order to use a different peer-to-peer

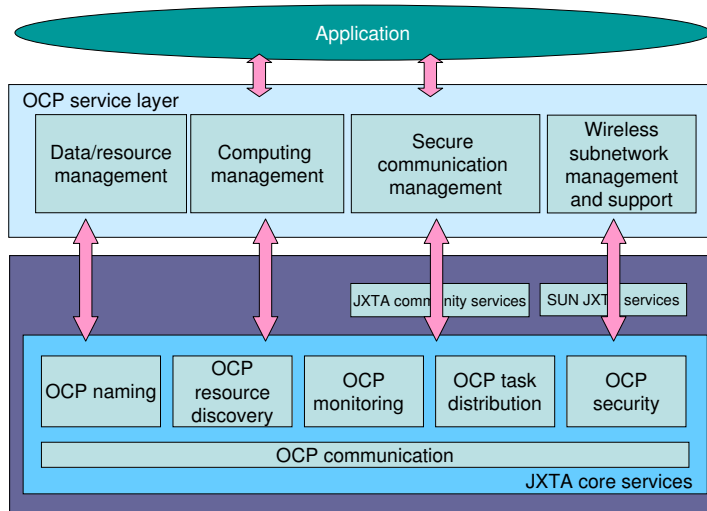


Figure 3: The Overlay Computing Platform software architecture: a flat organization.

overlay middleware.

A mixed approach, where some primitives are implemented at the underlying platform level (e.g., security primitives in JXTA) could be investigated in the development phase. As mentioned above, integrating some primitives such as security, communication or monitoring at the core layer will result in better performance, despite the heavy dependency on JXTA that this choice introduces. However, since performance of significant importance for the OCP, this is worth the drawback of a mixed architecture.

2.3 OCP functionalities

The functionalities supported by the OCP are organized in two layers (see Figures 3 and 4): the *OCP core layer* and the *OCP service layer*. The OCP core layer implements low level functionalities used by the OCP service layer to interact with the underlying platform and the peer-to-peer overlay network. The OCP service layer provides different programming primitives that can be classified as: data and storage management, computing management, secure communication management, wireless subnetwork management and support. In the following, we give an overview of both core layer and service layer functionalities; the service layer functionalities will be detailed in the remaining sections of this document.

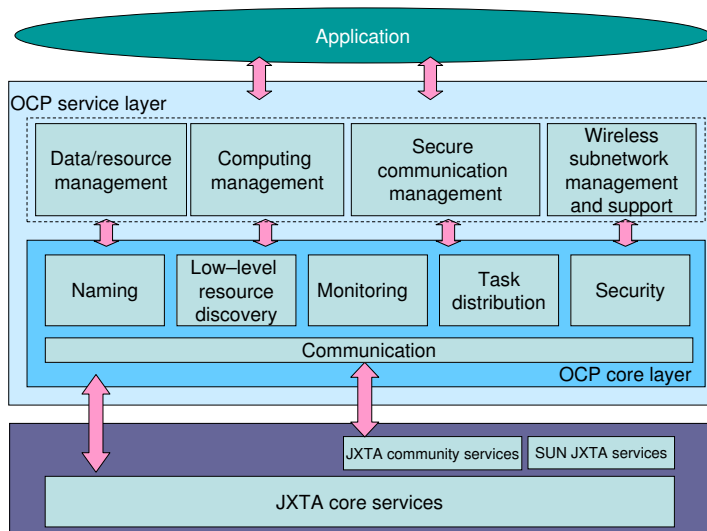


Figure 4: The Overlay Computing Platform software architecture: a structured organization.

2.3.1 The core layer

This layer provides primitives for communication, naming, low-level resource discovery, monitoring and security. These are the necessary primitives required by the service layer. However, since the architecture is open, it is possible to add more, if deemed necessary in the future. Although some of these primitives are provided as JXTA services, OCP will provide enhanced versions in order to support a rich set of low-level functionalities and guarantee a satisfactory performance level. Although, it is not the main priority of the OCP design, we will try to achieve the maximum possible degree of independence from JXTA.

Communication primitives will provide peer-to-peer and peer-to-group communication as well as inter-group and intra-group collective communication. Since JXTA provides the main communication primitives, we have decided to use these functionalities without build an OCP dedicated interface. However, the `net.jxta` package does not include collective communication primitives; these will be implemented as part of the OCP core layer (the communication block in Figure 3 and 4.) In summary, communication primitives provided by the OCP core include:

- peer group management primitives: these primitives allow to create a group of collaborating peers, to join a group and to leave a group.
- peer group communication primitives: these primitives allow the synchronous and asynchronous communication between peers of the same group or of different groups.

- peer group discovery primitives: these primitives allow to discover a group of peers (characterized by a JXTA ID), or a particular peer inside a specific group.
- collective communication primitives: these primitives allow to broadcast, multicast, anycast messages both among peers of the same group and among peers belonging to different groups.

Naming core primitives implement the naming functionality which associates a high level identifier to a JXTA ID. The idea is to provide a mapping of URIs, that identify a logical or physical resource, to JXTA IDs. In this way, the URI will be resolved into a *Codat*, i.e., into a combination of content and JXTA ID; these will be used in order to share content among peers. Typical URI schemes could be of the form `ocp:nodeName.DomainName/resourceName` or `ocp:resourceName.nodeName.DomainName` (corresponding instance examples could be `ocp:wlab1.ce.uniroma2.it/d321.doc`, and `ocp:d321.doc.wlab1.ce.uniroma2.it`, where the dots in the resource name are automatically translated to “_”).

The main set of naming primitives includes:

- `ocp.setName(lName, jxtaID)` associates the logical name `lName` to a resource (e.g., a peer or a service or any other logical resource).
- `ocp.deleteName(lName)` eliminates the `lName`, i.e., it destroys the mapping between `lName` and the associated JXTA ID but does not delete the JXTA resource or service.
- `jxtaID = ocp.getJxtaID(lName)` resolves an OCP `lName` into a `jxtaID`, if the mapping exists.
- `lName = ocp.getName(jxtaID)` resolves `jxtaID` into an OCP `lName`, if the mapping exists.
- `ocp.rename(lName1, lName2)` renames the resource `lName1` into `lName2`.

Low-level resource discovery primitives allow to find an OCP resource (logical or physical) which satisfies a set of properties. The main set of such primitives includes:

- `ocp.publish(resourceDescriptor)` allows to publish an OCP resource described by the `resourceDescriptor`. A resource descriptor is an XML document which extends the concept of the JXTA advertisements and which describes more properties of a resource such as: the hardware characteristics of a computing node, the computing platform provided by the node, the free space or the supported filesystem, the semantics of an OCP service, the guaranteed level of performance provided by an OCP service etc.
- `ress[] = ocp.search(resourceFilter)` allows to search for OCP resources having particular properties which match `resourceFilter`. The search will return a list of OCP URIs.

- `resDesc = ocp.getDescriptor(lName)` returns the descriptor of the resource `lName`.
- `ocp.delDescriptor(lName)` allows to delete an OCP resource descriptor.

Monitoring primitives will allow to monitor the status of OCP resources. The resource status is described by an XML document, delivered as an advertisement, and can be static or dynamic, depending on the resource and the metrics under consideration. The main set of such primitives includes:

- `mnr=ocp.createMonitor(monitorName, metrics[])` allows to create a monitor, uniquely identified by `monitorName`. The monitor will collect information specified by the vector `metrics[]`. The resource (physical or logical) that is being monitored is the resource that creates the monitor.
- `mnr.publishMonitoredData(metrics[], properties)` allows to publish monitored data, characterized by some properties. Properties allow to define different advertising policies, e.g., a limited time to live, a persistent data, advertising only the observations which differ from previous ones, etc. Published data are received by subscribers.
- `mnr.subscribeMonitoredData(metrics[], properties)` allows to receive monitored data that satisfy the specific properties by all monitors that have collected the specified metrics.
- `mnr.subscribeMonitor(monitorName, metrics[])` subscribes to a specific monitor, with the possibility to receive only a subset of the metrics collected by the monitor.
- `monitors[] = ocp.getMonitors(metrics[], properties)` returns a set of monitors collecting specific metrics and satisfying given advertising properties.

Task distribution primitives aim to support the high-level functionalities of the computing management service layer. They will be blended with monitoring primitives in order to efficiently provide computing functionalities such as performance-conscious allocation of distributed computations, efficient access to application servers, and Java thread migration. Primitives related to task distribution include:

- `clone()` creates a copy of the calling task; this copy inherits the original properties, data and state of the calling task.
- `activateTask(task, node)` launches a task at a computing node of the OCP. Together with the code to be run, `task` also includes the specification of input/output as well as settings for handling signals and permissions for interprocess communication.

- `migrateTask(task, node)` moves a task together with its current state to another OCP node.
- `signal(task, signaltype)` provides an asynchronous way to send signals of different types to tasks running on different OCP nodes. **Signal handlers** are responsible for the necessary actions when a signal is caught. Signals may be used in order to terminate some task abnormally.
- Primitives handling **semaphores** will provide low-level mechanisms in order to control access to low-level resources and provide task synchronization.

In addition, the core will contain basic security primitives, like encryption/decryption of communications, management of user/node credentials, e.g., digital certificates, (possibly transparent) authentication systems, basic trust negotiation mechanisms, secure distributed pseudo-random number generators and node signature primitives that will assist high level security functionalities of the service layer.

2.3.2 The service layer

In the following, we give an overview of the functionalities of the service layer; these functionalities will be detailed in the remaining sections of the document.

OCP service layer: Data/resource management. Data/resource management functionalities define storage primitives in order to allow access rights management, QoS requirements management (durability, availability and correctness), replication, naming and addressing, querying, indexing and distributed catalogues. Regarding replication, the functionalities will support automatic data placement (to fulfill QoS requirements) and automatic copy update and consistency management. This service layer relies on the naming and querying and discovery primitives of the OCP core layer.

OCP service layer: Computing management. Computing management functionalities will provide high level services and policies such as scheduling policies, performance-aware spawning and performance-oriented resource discovery. This service layer makes extensive use of the monitoring and task distribution primitives of the OCP core layer. Performance-aware spawning requires communication and computational performance measurements. Performance-oriented resource discovery is based on features such as computational capacity measurement and monitoring, point to point communication speed measurement and monitoring, global bandwidth measurement and monitoring.

OCP service layer: Secure communication management. Communication management functionalities will provide advanced security features such as anonymous communication, certified information access, reputation management for content filtering and

resource allocation, timestamping, secure function evaluation, etc. This service layer relies on security and communication primitives of the OCP core layer.

OCP service layer: Wireless subnetwork management and support. Wireless subnetwork management and support functionalities will allow OCP to integrate sensor networks. Functionalities of this service layer are classified in two levels. High level functionalities should support buffering, gateway, SQL-like queries, network statistics and control, virtual sensor networks, etc. Low level functionalities should support routing, mobility prediction, energy management, grouping, time synchronization, localization, etc. This service layer relies on communication primitives of the OCP core layer.

In the subsequent sections, we describe in detail OCP functionalities of the service layer and corresponding required low-level functionalities of the OCP core.

3 Functionalities for data/resource management

In this section we describe in an extensive way functionalities and corresponding API methods related to data and resource handling. These include functionalities that enable store, replication, indexing, naming, management of distribute catalogues, querying and groups and resource addressing.

3.1 Store service

Description: This service will be a basic primitive in the OCP that will be used for both storing the data in the OCP's virtual file system and for information sharing applications. At its core, this primitive allows the user/programmer to physically store his data at remote nodes that run the OCP. The programmer is not aware of where his data is stored but he still must be able to locate it whenever required (through lookup and indexing). When the store data functionality is called, the system discovers some nodes that it considers appropriate (through resource discovery) and stores the data there.

The store data functionality, besides just physically storing data at remote nodes, also makes it available to all the users of the OCP that may or may not access it depending on the data *access rights*. These access rights are defined by their owner, allowing him to make his data available publicly to all users, a group of users (defined by a set of credentials) or just himself.

Besides access rights, the programmer may specify storage-related data quality guarantees. Such guarantees include *durability*, *availability* and *correctness*. Durability ensures that the stored data will survive network and site failures. Availability ensures that data will be available for access anytime and anyplace, despite transient failures, traffic bursts or other performance impediments. There may be various degrees of availability, specified for instance in terms of time (e.g., stored data may be available 90% of the time), space, or means of access (i.e., stored data are not available for access from a cellular phone). Correctness refers to issues related to data updates. First, it ensures that data may be concurrently accessed by many users. Then, in case of replication, it provides guarantees about *freshness*. There are many ways to express freshness such as *lag-based* (limits the number of consequent updates that are not applied to a copy), *divergence-based* (bounds the divergence between the value of any copy and the value of the most current one) or *time-based* (specifies the maximum acceptable distance in time between the time the copy is created and current time).

API:

- Typical primitive functionalities for handling data include `CreateData`, `RemoveData`, `UpdateData` and `ReadData`
- Access rights are managed using `DefineAccessRights` and `UpdateAccessRights`

- Data quality guarantees associated with data/resources are defined using `DefineDataQuality` and `UpdateDataQuality`

Related functionalities: Indexing service, Certified information access, Replication service

3.2 Replication service

Description: Besides a separate tool to the programmer, the replication service will be transparently used by the OCP in order to guarantee the availability of the store service. The degree of replication per file may either be specified by the programmer or it must be such that the specified availability is attained. The OCP deploys appropriate data placement and update protocols in order to ensure the data quality guarantees specified. The replication degree is also related to the popularity of the file and may be adapted dynamically. When calling the replication service explicitly, the programmer must provide a protocol for the creation and placement of copies. Finally, the programmer also specifies a protocol for updating the copies. The update protocol must be such that the specified freshness guarantees are satisfied.

API:

- `CreateReplicas`: Creates replicas of a file and distributes them to remote OCP nodes according either to explicit replication degree or to data quality guarantees specified (availability, popularity)
- `UpdateReplicas`: Defines a protocol for updating copies of a file. It receives as parameter the required freshness guarantee of the file

Related functionality: Store service

3.3 Indexing service

Description: Every piece of data stored in the OCP will be indexed so it can be efficiently retrieved. The system will provide the basic functionality for indexing data. In particular, whenever the store data functionality is called, it triggers the execution of the indexing primitive that inserts the newly stored data into the OCP's indexing infrastructure. The index is looked up whenever a user needs to access a piece of data.

The indexing functionality has to distinguish between typical raw-data files and information (i.e., data files with semantic meaning). For the first type of data, the system should support a basic indexing technique based on some naming conventions. Thus, a user locates this type of data by using only its name as a virtual identifier. The functionality of indexing information is for example used for file sharing and distributed document management applications where the users need to locate and retrieve data based on their

content and not just their name. In this case, the index is built on a semantic description of the data, i.e., its metadata. However, this semantic description depends both on the type and content of the data and also on the application that uses the data. Thus, a single data item (file) may have multiple semantic descriptions, one for each of the applications that uses it. This description may be just its filename or some keywords in its simplest form or more sophisticated descriptions such as XML or RDF descriptions of the content of the files. The system will provide some basic indexing functionality on the metadata description that could be replaced by an application-specific indexing.

An index may be built for both data (files) and resources (such as nodes, computational or storage resources). Indexes for resources may be based on the name or type of the resources or in other characteristics such as the current resource utilization or load. We do not differentiate between index functionality for data and index functionality for resources. However, to maintain an index with information about the current status of the resource, a resource monitoring functionality may be needed. In particular, each resource/node/entity in the system should be described with some characteristics. Some of these characteristics are used by the OCP to determine which nodes are suitable for storing data from other nodes (store data functionality), for executing distributed computations (task allocation) and for other operations of the system, such as load balancing. These characteristics can be divided into static characteristics such as CPU type, dynamic characteristics such as current load, available space/memory and complex characteristics such as latency and bandwidth that are not determined by a single node (for example, one needs to know at least two nodes to measure the latency between them). These characteristics are derived from the system and expressed using XML or RDF.

Part of the indexing service is a lookup operation to efficiently locate data stored in the OCP or resources available by OCP through exploiting the available indexing infrastructure. The lookup functionality enables the programmer to locate them through some key (e.g., the filename that the naming functionality has assigned to the stored data or the metadata description for more complex information resources). Thus, the system should support any query that can be posed on the contents of these descriptions including range queries on attribute-value pairs, conjunctions and disjunctions of keywords, path queries for XML descriptions, etc.

API:

- **CreateIndex:** Creates an index on a set of data/resources
- **UpdateIndex:** Updates an existing index
- **InsertToIndex:** Inserts a new key to an index
- **DeleteFromIndex:** Deletes a key from an index
- **Lookup(key):** Searches for data/resources matching a given key

- **GetResourceCharacteristic:** Returns the characteristics of a computational or communication resource

Related functionalities: Store service, Naming service, Replication service, Query service

3.4 Naming service

Description: To implement the indexing service for the data, some naming conventions or naming functionalities are required. This naming scheme will be global in the OCP and will be generalized to providing system-wide names for resources or data items in general.

API:

- **NewResourceIdentifier:** Creates a new (randomly generated) identifier for resources
- **AssociateResourceIdentifier:** Associates an identifier to a resource

Related functionalities: Store service, Indexing service, Distributed catalogue service, Query service

3.5 Distributed catalogue service

Description: Besides the various indexes that may be available on the data and resources of the OCP, a distributed directory or catalogue service will be needed in order to provide information about the general configuration of the system. The catalogue will include statistics about resource utilization, distribution of resources and data, load conditions or availability of indexes. It may be possible to have more than one catalogues, for instance, there may be one catalogue per group.

API:

- Typical operations include **CreateDirectory**, **DeleteDirectory**, **UpdateDirectory**, **InsertToDirectory**, **DeleteFromDirectory** and **LookupDirectory**.

Related functionalities: Naming service, Indexing service, Store service, Replication service

3.6 Query service

Description: OCP will provide the means for querying for data and resources. The query language may be some declarative database inspired language such as SQL or XQuery. The query service assumes the existence of a query processor and a query engine

that uses catalogues and indexes. An SQL query engine should support simple relational operations, such as select and join, as well as aggregation operations. An XQuery query engine should support at least XPATH queries.

API:

- **SQLQuery:** Performs Select-Project-Join-GroupBy queries on a set of data. It may take as parameter an index and a set of data.
- **XPATHQuery:** Performs XPATH queries on XML documents

Related functionalities: Store service, Indexing service, Naming service, Distributed catalogue service

3.7 Groups and resource addressing

Description: OCP will allow nodes to form groups. The formation of groups may be viewed as a more sophisticated node join functionality. Each node, when entering the OCP, does not just join the overlay computer but it may also choose to join one or more particular groups. This can be also done while a node is already part of OCP. Groups can be formed based on the content of the nodes. Upon entering the system, the node provides its content description (e.g., a concise metadata description of its contents) and the system determines the group with which it has the most similar contents so as to attach to it. Groups can be also based on other properties such as topological criteria, security or administration policies. Finally, groups may be divided into subgroups, forming group hierarchies through split and merge operations. The Group service is expected to be based on the corresponding group primitive service of JXTA.

Mechanics: We allow nodes of the OCP to associate a resource (e.g., a file or a task) to an identifier within a group. The addressing within a group will use a unique hash function that will presumably be negotiated at group formation. Nodes joining the particular group of the OCP adopt the chosen hash function. This functionality will resemble DHT. We plan to use JXTA's DHT functionalities and further enhance them.

API:

- Typical related operations include **CreateGroup**, **DeleteGroup**, **UpdateGroup**, **Merge Group/Split Group**, **JoinGroup** and **LeaveGroup**
- **GetHashFunction(GroupID):** Invoked by a node joining a group, this method returns the hash function adopted by group identified by **GroupID** when joining it.

- `GetAddress(Resource, HashFunction, GroupID)`: This method returns the node ID (belonging to group `GroupID`) associated to `Resource`, computed using `HashFunction`. `GroupID` is necessary, since a node can belong to several groups. The `HashFunction` argument can be used for error checking, i.e., to match a hash function to the correct group.

Implementation notes: The basic idea is that many applications (such as distributed file sharing) are group based. This can drive the negotiation of the hash function. A first approach can be that the node that initiates a group chooses the hash function for the group and all nodes successively joining the group are bound to this choice. Note that `GetAddress` has to return the node ID of an active node within the group.

3.8 Summary

The next table summarizes the main functionalities and API methods described in this section.

Functionalities	API methods
Store service	<code>CreateData</code> , <code>RemoveData</code> , <code>UpdateData</code> , <code>ReadData</code> <code>DefineAccessRights</code> , <code>UpdateAccessRights</code>
Replication service	<code>CreateReplicas</code> , <code>UpdateReplicas</code>
Indexing service	<code>DeleteFromIndex</code> , <code>UpdateIndex</code> , <code>InsertToIndex</code> <code>CreateIndex</code> , <code>Lookup</code> , <code>GetResourceCharacteristic</code>
Naming service	<code>NewResourceIdentifier</code> , <code>AssociateResourceIdentifier</code>
Distributed catalogue service	<code>CreateDirectory</code> , <code>DeleteDirectory</code> , <code>InsertToDirectory</code> <code>UpdateDirectory</code> , <code>DeleteFrom Directory</code> , <code>LookupDirectory</code>
Query service	<code>SQLQuery</code> , <code>XPATHQuery</code>
Groups and resource addressing	<code>CreateGroup</code> , <code>DeleteGroup</code> , <code>UpdateGroup</code> <code>MergeGroup</code> , <code>SplitGroup</code> , <code>JoinGroup</code> <code>LeaveGroup</code> , <code>GetHashFunction</code> , <code>GetAddress</code>

4 Functionalities for communication and distributed computations

In this section, we first describe some basic communication functionalities, which rely on JXTA pipes, that the OCP will provide and then focus on functionalities that implement schedulers and enable performance-conscious allocation of distributed computations, as well as JVM farm management and thread migration.

4.1 Basic communication primitives

Description: Basic communication primitives of the OCP will be based on *pipes* which is the basic communication mechanism provided by JXTA for communication. Essentially, JXTA pipes implement an asynchronous and unidirectional message transfer mechanism. A pipe is a virtual channel and the endpoints of a pipe may be bound to one or more node (peer) endpoints. More complex communication primitives are obtained by combining input and output pipes.

We plan to enhance this mechanism in order to provide a set of necessary communication functionalities to the OCP and realize several communication patterns such as point-to-point (or unicast), propagate (or one-to-many) including broadcasting and multicasting, gathering (e.g., in order to support prefix sum computations), and many-to-many (including all-to-all) communication within a group of nodes. We will develop dynamic pipes that will allow nodes to join pipes after their original creation and support many pipes of different types simultaneously. Furthermore, our advanced pipes will have additional desired properties such as guaranteed levels of reliability, security, and quality of service (e.g., guaranteed transfer delay).

JXTA pipes enable communication between nodes that are not directly connected and may belong to different subnetworks, separated by firewalls and NATs. This is already provided by the endpoint protocol in JXTA. We plan to enhance the endpoint protocol by providing more sophisticated routing protocols over nodes (peers) and incorporating them into the OCP.

API:

- Typical functionalities for handling pipes include `CreatePipe`, `OpenPipe`, `ListenPipe`, and `ClosePipe`. Parameters such as indicators whether the pipe supports secure transfer, protection requirements (for fault-tolerance), bandwidth/delay requirements (as QoS guarantees), etc. will also be supported.
- Simple functionalities such as `send/receive` as well as more structured ones such as `Broadcast/Multicast/GatherInfo` within nodes of different groups will be provided for message transfer through pipes.

- Routing protocols will provide interfaces to the programmer/user such as `FindRoute`, `FindTree`, etc. These functionalities will return routes or communication patterns meeting specific criteria set as parameters including encryption requirements and QoS guarantees.

Related functionalities: Secure broadcast, Performance-conscious allocation of distributed computations

4.2 Schedulers for application servers

Description: The aim of this functionality is to support the scheduling decisions of a server or a set of servers that provide some service in applications. It is a typical situation when a server provides some service to clients but due to several computation/communication constraints on the server side or due to parameters of requests (jobs) only a few number of requests can be efficiently satisfied or requests can only be satisfied at different performance levels.

The scheduler does not intend to satisfy the users' requests; this task is performed by the server(s). Instead, the scheduler makes the scheduling decision based on (estimates about) the performance of servers, the parameters of requests, and the scheduling objective, i.e., it is responsible for computing the schedule that minimizes some objective.

Mechanics: A large variety of scheduling policies will be available. These include scheduling on a single server or on multiple servers, jobs with release times and deadlines, single jobs or structured jobs with precedence constraints, unreliable servers, jobs with variable service time and more. These parameters are defined when a scheduler is launched or when new requests (jobs) appear.

The scheduler may run independently of the server(s) if the parameters of the scheduling problem are well-defined or may interact with the server(s) in order to adapt to changes, update the estimates, and get feedback on the effect of scheduling decisions (e.g., what was the actual service time for a job on which only a probabilistic assumption had been considered in order to make the scheduling decision).

API:

- **StartScheduler:** Initiates a scheduler by specifying the parameters of the server(s). Server parameters include their number, speed, probability of availability, objectives.
- **StopScheduler:** Stops the scheduler.
- **RecommendDecision:** The scheduler recommends a decision to a server.
- **ReportSchedule:** Returns a report on the scheduling decisions for a period of time.

- **ServerEvent:** A server or a monitor associated with a server informs the scheduler about an event on the side of the server. Such events include the beginning or the termination of a job, a fault of the server (such events are issued by monitors), or some change in service such as the addition of a new server, the unavailability of a server, the change in its parameters (increase/decrease of speed). Furthermore, this is the typical response of a server to a `RecommendDecision` command by the scheduler.
- **JobRequest:** Denotes the request for the beginning of a new job or the termination of an already scheduled job.

Implementation notes: The definition of the scheduler should take into account different characteristics of the server(s). Typical such characteristics include the number of servers, their availability, their speed, the communication bandwidth, and the objective that has to be fulfilled. The objective may be server-centered (e.g., total service time) or client-centered (e.g., probability of accessing the server, service time, avoidance of starvation after continuous requests, etc.).

The server may be dedicated to handling requests or this service may have very low priority. The server itself may pose restrictions to the scheduler and the jobs that wish to be scheduled. For example, in many cases, it is reasonable to assume that the load of all jobs is identical; hence, the service time depends linearly on the speed of a server.

This functionality strongly cooperates with the monitoring primitives of the OCP core layer.

Related functionality: Performance-conscious allocation of distributed computations

4.3 Performance-conscious allocation of distributed computations

OCP will provide functionalities to make the computational power of the participating nodes available for the execution of parallel/distributed applications. Most of the current parallel applications running on peer-to-peer systems are computationally intensive (e.g., SETI@HOME) and are characterized by a computational structure where little to no communication is required among the single component processes. Often, the computational graph of these applications reduces to a single master node allocating (possibly replicated) chunks of computation to a number of clients, and then gathering the results.

When running more complex applications with a certain degree of interaction between the participating entities, one of the challenges is to guarantee that communication performance is sufficient to obtain a reasonable speed-up, so that resorting to a distributed execution is meaningful in the first place.

For this purpose, we envision the need of a spawning functionality which is able to allocate a distributed computation to a subset of nodes guaranteeing some (perhaps in-

evitably rough) measures of performance. Possible parameters for the spawning primitive are:

- the (SPMD) code to be executed by the parallel processes,
- the degree n of parallelism required by the application,
- the degree r of redundancy for each task (for fault tolerance). The number of nodes to be allocated will then be $p = n * r$,
- the minimum computational power (e.g., in FLOPS or MIPS) that each node involved ought to guarantee,
- the maximum latency tolerable for each communication and
- a minimum requirement of global bandwidth (e.g., bisection bandwidth).

Parameters corresponding to resources that are not critical or for which service level guarantees cannot be enforced, could be set to default values and ignored. Once a suitable subnet of nodes is assigned, the ensuing computation can then be efficiently implemented using communication libraries built on top of the global computer. Efforts in this direction are already ongoing in the scientific community. An interesting example is the P2P-MPI project, which aims at porting the popular MPI message-passing primitives onto desktop grids.

The implementation of the spawning functionality will require consideration of other issues such as dynamicity, selfishness, and fault tolerance, which are of interest within AEOLUS.

Clearly, spawning is a complex functionality, which must rely on a number of lower-level primitives instrumental in locating the configuration of nodes meeting the desired characteristics. A possible list of “system functionalities” is discussed below.

4.3.1 Discovery of performance characteristics

Primitives for probing the OCP nodes to gain trusted information about their computation and communication performance are needed to implement the spawning functionality. Different types of probes are described below:

- a. *Measures of computation speed:* To allow for a trustable measure of computation speed, a host could let the probed node to produce a short answer which necessarily requires a given amount of computation to be performed. Then, comparing the time needed to produce the answer with the ping time may be used to extrapolate the actual computing power available at the remote node. An instance of such a function could be a simple random number generator, where the probing host sends the seed and the probed node returns the n -th number generated, where n is a parameter of the probing procedure. If the answer is not correct, the probing host ignores the answer and may involve the reputation facility to signal this anomaly.

- b. *Measures of point-to-point communication speed:* A ping-like ability should be enough to obtain a rough estimate of point-to-point communication speed. In fact, this measure is mostly related to latency, but it may give a first approximation also for bandwidth. To prevent that the probed host maliciously delays the ping acknowledgment, suitable rewarding/reputation mechanisms should be enforced to discourage/avoid potentially unfair behavior of hosts.
- c. *Measures of global bandwidth:* The most delicate aspect in selecting an appropriate set of nodes to perform the parallel computation is to guarantee that the chosen subsystem exhibits enough communication bandwidth. Measuring bandwidth is not a simple task. In fact, its very definition depends on the actual computation to be performed. For instance, for any tree-like computation (e.g., a distributed branch-and-bound activity) it suffices to make sure that the point-to-point communication channels between each parent and child nodes achieve the required threshold. More complex parallel computations (e.g., large-scale FFT computations) require a lower threshold on the bisection bandwidth of the system. Bisection bandwidth of the spawned subsystem could be probed by performing a number of (random) data exchanges among the involved nodes. Most likely, however, applications which are likely to be programmed on the OCP will exhibit a very sparse communication graph, therefore, the point-to-point measures described in Point b) should be sufficient to guarantee a reasonable level of performance.

In general, the primitives implementing the above measurements will make use of suitably designed microbenchmarks which run on candidate nodes and gather the required information.

4.3.2 Using local non-busy cycles to gain performance information

The spawning strategy for computational tasks could benefit from a suitable combination of information gathered by light-weight processes continuously running in the background (learning the long term performance trends) and information obtained by a more intensive probing made just before the spawning must occur (learning the short term properties). The tradeoff between *long term* and *short term* information in driving decisions depends on the dynamicity of the environment. Some works in the literature seem to suggest that knowledge of long term behavior could improve performance even in a very dynamic setting. Hence, a functionality devoted to probing and learning should be considered when designing the OCP.

More specifically, hosts could use a share of their free cycles to probe the communication and computation capability/availability within the OCP, with the primitives sketched above, to gain knowledge of the system. The relevant information could be kept in a table of other hosts with trusted/stable computation/communication capabilities that could statically drive the embedding of the parallel application. Whether sharing this table

would bring advantages is an issue for investigation.

4.4 Implementation of distributed JVM for thread migration

Description: Java language and JVM features motivate the efforts made by researchers on studying an effective and efficient distribution of Java applications among a set of computing nodes. In this context the Distributed Java Virtual Machines (DJVM) is one of the most important research areas, where the focus is on java thread migration over clustered environments for balancing the workload. The clustered computing environment is not very scalable and basically requires homogeneity of nodes.

We plan to provide, especially for java multithreading applications which make heavy use of the threads, a highly scalable and heterogeneity-unaware computing environment that offers functionalities for building as large as possible JVM farms. We foresee a layer offering functionalities related to:

- collecting information about other available sites,
- building/managing farms (initializing a new farm and/or joining an existing one),
- monitoring the workload of nodes participating in the computation,
- the migration of local threads,
- retrieving migrated threads,
- managing the synchronization concerns (e.g., thread context), etc.

Implementation notes: The above functionalities can be built over JXTA. In particular, the groups mechanism can be extended in order to initialize and join JVM farms. Query/response advertisements can be used in order to discover the topology of nodes and collect information about them and their services. The *Peer Information Protocol* can be used to implement methods for monitoring and synchronizing the farm. Relay and pipes mechanisms can also support thread migration in multithread applications.

Related functionality: Performance-conscious allocation of distributed computations

4.5 Summary

The next table summarizes the main functionalities and API methods described in this section.

Functionalities	API methods
Communication	CreatePipe, OpenPipe, ListenPipe, Send Receive, ClosePipe, Broadcast, FindTree Multicast, GatherInfo, FindRoute
Schedulers for application servers	StartScheduler, StopScheduler, ServerEvent ReportSchedule, RecommendDecision, JobRequest
Performance-conscious allocation of distributed computations	Spawn, Probe
Thread migration	MigrateLocalThreads, RetrieveLocalThreads SyncManagement, MonitorLoad, ManageFarm

5 Functionalities for security and trust management

In this section we focus on functionalities related to security and trust issues that are being addressed in the context of AEOLUS. We describe the relevant functionalities, sketch the corresponding API and briefly introduce some basic API methods, as well as comment on implementation issues.

5.1 Anonymous communication network

Description: Much information can be derived from the analysis of communication patterns of users. If the communication is traceable, an adversary capable of deploying traffic analysis attacks is able to break anonymity mechanisms implemented at the data level (e.g., anonymous or pseudonymous identity management mechanisms) and identify the users' actions. Anonymous communication networks make it difficult for an adversary to trace the communication of a user. This is achieved by using intermediate entities in the routing that relay the communication for the user.

Mechanics: In a distributed architecture such as the OCP, anonymous communication networks could be implemented in the form of an anonymous P2P network (i.e., users themselves provide the anonymous service to each other). Instead of sending requests or information directly to its destination, users create a path through other entities in the OCP. Linking the source and destination of the communication becomes hard for adversaries.

API:

- `CreateAnonymousChannel(destination)`: Creates an anonymous channel from the source of the communication to the destination. This primitive may involve several intermediate steps (e.g., contacting a few entities)
- `SendInformation(channel)`: Send information anonymously through the channel
- `CloseChannel(channel)`: Close the anonymous channel once it is no longer needed
- `RelayInformation()`: Users will be asked to relay information for others. They act not only as potential senders of information but also as anonymity providers.
- `AnonymousReply(request)`: Users may want to be able to reply to anonymous senders. Anonymous reply mechanisms need to be implemented.

5.2 Certified information access

Description: This functionality involves three parties: the `CertifiedDB`, the `User` and the `PublicInformationStorage`. This functionality allows one party (the `CertifiedDB`) to

commit to some piece of information and answer queries (issued by a user) about the committed information. The query replies carry a proof of validity that can be verified by the user by using some public information that the `CertifiedDB` publishes (using a `PublicInformationStorage`). The public information should not reveal anything about the actual content of the database and the query replies (and the proofs of validity) should only reveal the result of the query and no additional information.

Mechanics: The `CertifiedDB`, based on the content of the database, produces some information that is then sent to the `PublicInformationStorage`. The `PublicInformationStorage` associates a unique Id to this information. Whenever a user makes a query to the database, he obtains an object that contains the answer to the query and some information that can be used, along with the information held by the `PublicInformationStorage`, to prove that the answer is indeed correct and that the database has not cheated.

API:

- `CertifiedDB`
 - `generatePublicInfo`: Provides the `CertifiedDB` with the public information that has to be sent to the `PublicInformationStorage`.
 - `sendPublicInfo`: Allows the `CertifiedDB` to transfer the public information to a `PublicInformationStorage`.
 - `deletePublicInfo`: Allows the `CertifiedDB` to require the removal of the public information from the `PublicInformationStorage`.
 - `getId`: The `CertifiedDB` obtains a unique identifier from the `PublicInformationStorage`.
 - `sendId`: The `CertifiedDB` provides a user with the information needed to identify the public information related to its database. In particular, it provides the list of `PublicInformationStorage` and the associated Id to its local information.
 - `certifiedAnswer`: Given a query, construct an object `CertifiedAnswer` that contains the answer to the query along with its proof of validity.
 - `sendAnswer`: Sends a `CertifiedAnswer` to the user.
- `User`
 - `getPublicInfoId`: The user obtains from the `CertifiedDB` the location and the Id of the public information.
 - `getPublicInfo`: The user obtains the from the `PublicInformationStorage` the public info related to a `CertifiedDB`.
 - `certifiedQuery`: The user issues a query to the database.

- `getCertifiedAnswer`: The user obtains a `CertifiedAnswer` to its query.
 - `verifyAnswer`: The user verifies the validity of the information contained in the `CertifiedAnswer` by checking the proof therein against the public information.
- `PublicInformationStorage`
 - `getInfo`: The `PublicInformationStorage` obtains the public information from a `CertifiedDB`.
 - `deleteInfo`: The `PublicInformationStorage` deletes the public information for a specific `CertifiedDB`.
 - `sendInfo`: Provides the user with the public info associated to a specific Id.
 - `generateId`: This method allows the `PublicInformationStorage` to associate a unique Id to the public information received from a `CertifiedDB`.
 - `sendId`: The `PublicInformationStorage` send to the `CertifiedDB` the Id related to its public information

5.3 Reputation management for content filtering

Description: Some of the applications for the overlay computer will allow any users to perform actions (e.g., posting articles, comments or updating a wikipedia article). These facilities can be abused persistently by certain users, e.g., they spam, insult, post inaccurate/irrelevant stuff etc. Even in large distributed systems it is difficult to filter those abusers out based on their “identity”: they have the ability to change their identity often (these are called sybil attacks: change of IPs, re-registering under different names, change of email address), as soon as their posts start getting filtered out.

Mechanics: Instead of requiring users to simply register a fresh pseudonym to use a service, we also require them to be “introduced by” an existing user of the system. Then, their name is then not simply the pseudonym but the whole chain of names from the administrator of the system on which the action is performed to that pseudonym. An adversary finds it now difficult to change their identity completely, since they would have to convince many different people to introduce them into the network. This is not easy since others might be reluctant to introduce those that they do not trust, as if they were to be abusers the introducer’s name will also be linked with the abuse.

Based on the previous simple introduction mechanism we then develop a policy about labeling actions (such as articles or wikipedia changes) that allows users to “object” to actions, and to “take responsibility” for actions. This allows for actions to be tagged with persistent labels that can be used for filtering by end users.

API:

- Actions performed by the users forming the network

- `Introduce(user)`: Introduce a new user in the reputation system
 - `Label(action)`: Attach a label to an action (article, post, change)
 - `Object(action)`: Present an objection to an action (spam, defamatory, irrelevant, etc.)
 - `Object(objection)`: Present an objection to an objection
 - `TakeResponsibility(action)`: take responsibility for a certain objected action
 - `RevealPath`: In the case that no user takes responsibility, the first user on the path from `Root` to `Alice` is asked for the identity of the user that connects them to `Alice`. If that user fails to comply, then they are automatically assumed to be taking responsibility for the controversial post, and their full identity is associated with it. Thus, users that connect `Alice` to `Root` start revealing the path to `Alice`, unless they accept responsibility for the post.
 - `FilterOut(labels)`: Filter out actions for which an objection has been presented by some user(s) or those for whom some user(s) have taken responsibility.
- Actions performed by the central server

The central server will respond to the calls from users (e.g., introduce new user, check that actions have correct labels, handle objections and responsibilities).

- `CallForResponsibility(action)`: When an objection has been presented, the central server launches a call (broadcast) to all users to ask the author to take responsibility for an action that has been objected. If no user takes responsibility for it, the path from `Root` to the author starts to be revealed, until somebody takes responsibility for the action. This process is handled by the central server.

Implementation notes: The reputation system for content filtering can be realized in different ways. The simplest implementation would involve one trusted central server that takes care of all interactions with users (introductions, registrations, actions, objections and taking responsibility). That should be implementable as a set of web services, that other distributed applications could use. This is the best place to start and would give a lot of value to distributed nodes in reducing the abusive actions they are subject to. This pilot implementation in fact would test whether the policy is indeed effective in reducing abuse – which is its key property.

The second option is to distribute the functionality on many servers. They may all be trusted not to cheat, but also are not automatically collaborating with each other. This may start providing users with some of the privacy properties, that are the second issue and property of the protocol.

As a third option, we could aim for a completely peer-to-peer and decentralized system, where each user runs a program that takes care of interactions with the system. The (theoretical) fully distributed solution we have is very expensive (in terms of interactions and crypto) and quite fragile (it was only meant as a proof of concept and is quite open to attack). So this last part is still a research problem, we would not favor it just yet for the first pilot implementation.

5.4 Reputation management for resource allocation

Description: Reputation management involves recording a person’s or agent’s actions and the opinions of others about those actions. These records can then be published in order to allow other people (or agents) to make informed decisions about whether to trust that person or not. An example is the feedback system on eBay where each user posts their opinion (positive or negative) on the person they transacted with. Since having primarily positive feedback will improve a user’s reputation and therefore make other users more comfortable in dealing with him, users are encouraged to behave in a fashion beneficial to the person they are transacting with.

A reputation management system which uses pre-programmed criteria for reputation management automates the process of encouraging cooperative behavior over selfish behavior. In the context of the OCP, reputation management is needed for a fair resource allocation. Users should be encouraged to share their resources and “free riders” should be prevented from abusing the system.

Mechanics: The key idea for a reputation management system for resource allocation is that users of the OCP earn reputation when they share resources (computational power and memory space). They can later use this reputation to ask for resources from other users.

When user Alice asks user Bob for a resource (e.g., microprocessor cycles or memory space), Bob can check if Alice is a cooperative user who shares her own microprocessor and disk space when required by others. Alice shows Bob her reputation status, and Bob decides whether to grant Alice or not the resource based on her reputation. At the end of the transaction, Alice updates Bob’s reputation: it will be increased if Alice is satisfied with the transaction, and decrease otherwise.

The system should be secure against users tampering with their reputation or cheating in any way. Designing a scheme fit for securely managing reputation for resource allocation in the overlay computer in a fully distributed fashion is an open problem.

API:

- `getReputation(user)`: Used to get the reputation of a user
- `updateReputation(user)`: Update the reputation of a user after a transaction has taken place

- `showReputation`: Show the reputation status to another user
- `Decide(reputation)`: Decide whether or not to grant a user access to a resource given his reputation.

Implementation notes: Designing a reputation management system suited for the OCP resource allocation, that is fully distributed, secure against abuse and efficient is an open question.

5.5 Secure broadcast (Byzantine agreement)

Description: Byzantine agreement is a kind of guaranteed multicast of a value in which one process sends its value to other participants and they exchange various messages in order to agree on exactly what value was sent. It results when all correctly operating processes are able to agree either on a value or on the conclusion that the originator of the value is faulty. It is called Byzantine because we make no assumption about the behaviour of any undetected faulty processes. More explicitly, Byzantine Agreement is achieved when:

- (I) all correctly operating processes agree on the same value, and
- (II) if the transmitter operates correctly, then all correctly operating processes agree on its value.

Implicit in (I) and (II) is the idea that the agreement is synchronous in the sense that all processes reach this agreement at the same time. In other words, there must be some real time at which each of the processes has completed the execution of its algorithm for reaching agreement, and this time must be known in advance.

Mechanics: It is assumed that a group of participants is set and that the goal is to securely broadcast a message to all members of this group. Thus, every participant is able to send (broadcast) a message and he is constantly listening for upcoming messages.

API:

- Users
 - `BroadcastMessage`: The user sends the intended message
 - `ReceiveMessage`: This is counterpart of the `BroadcastMessage`

Implementation notes: The implementation of this functionality may use `TrustedThirdParty` which receives the message send by the user and then sends to all (the rest of the) participants.

5.6 Secure function evaluation with limited trusted third party

Description: Secure function evaluation primitives allow a set of n users to compute the value of a given function f given their inputs in such a way that at the end of the computation each user knows the value of the output but has no information about the value of the input of other players. In the more general case, the function f may associate to each participant a different output. Furthermore, one of the participants may initiate the protocol by sending to all the other participants a description of the function. We consider the case in which users obtain a certified description for a given function from a trusted service. Furthermore, we assume that a trusted third party has a minimum participation in the protocol in order either to reduce the complexity or to overcome some impossibility result. A function in this case will be a pair composed by the specification of the function for the users and the one for the trusted third party that, in principle, may be different.

Mechanics: In an initial phase, the participants and the trusted third party obtain the function description from a `TrustedFunctionRepository`. If the protocol requires it, the users execute a protocol that assigns to each user a identity, e.g., in the range $1..n$. This protocol also identifies the peer that acts as `TrustedThirdParty`. In a second stage, each participant will translate the function into a suitable representation. Such representations may be subject to verification by the other peers. At this point the actual function evaluation protocol is executed by all the participants.

API:

- `TrustedFunctionRepository`
 - `SendFunction`: `TrustedFunctionRepository` sends the description of the function to the users.
 - `SendTTPFunction`: `TrustedFunctionRepository` sends the description of the function for the trusted third party to the `TrustedThirdParty`.
- `TrustedThirdParty`
 - `ReceiveTTPFunction`: This is the counterpart of the `SendTTPFunction`.
 - `TranslateTTPFunction`: The function is translated into a proper representation (e.g., a boolean circuit), suitable for the evaluation process.
 - `FunctionEvaluation`: This method evaluates the value of the function (e.g., GMW).
- Users

- **DefineIdentities:** This method associates to a user a specific identity that will be used during the execution of the protocol.
- **ReceiveFunction:** This is the counterpart of the **SendFunction**.
- **IdentifyTTP:** Provides the users with a method to identify another peer as **TrustedThirdParty**.
- **TranslateFunction:** The function is translated into a proper representation (e.g., a boolean circuit), suitable for the evaluation process.
- **PrepareInput:** This method converts the local input of the participant in a suitable way for being used during the evaluation.
- **VerifyRepresentation:** Depending on the specific protocol, the participants may be required to exchange the new representation of the function. In this case, this method needs to implement a verification procedure that checks whether the representation sent by another participant represents the original function.
- **FunctionEvaluation:** This method evaluates the value of the function (e.g., GMW).

Implementation notes: The implementation of this functionality may require the specification of an ad-hoc language that should be used to specify the function to be computed and the function implemented by the **TrustedThirdParty**.

5.7 Secure function evaluation

Description: Secure function evaluation primitives allow a set of n users to compute the value of a given function f given their inputs in such a way that, at the end of the computation, each user knows the value of the output but has no information about the value of the input of other players. In the more general case, the function f may associate to each participant a different output. Furthermore, one of the participants may initiate the protocol by sending to all the other participants a description of the function. We consider the case in which users obtain a certified description for a given function from a trusted service.

Mechanics: In an initial phase, the participants will obtain the function description from a **TrustedFunctionRepository**. If the protocol requires it, the users execute a protocol that assigns to each user a identity, e.g., in the range $1\dots n$. In a second stage, each participant will translate the function into a suitable representation. Such representations may be subject to verification by the other peers. At this point the actual function evaluation protocol is executed by all the participants.

API:

- **TrustedFunctionRepository**
 - **SendFunction**: **TrustedFunctionRepository** sends the description of the function to the users.
- **Users**
 - **DefineIdentities**: This method associates to a user a specific identity that will be used during the execution of the protocol.
 - **ReceiveFunction**: This is the counterpart of the **SendFunction**.
 - **TranslateFunction**: The function is translated into a proper representation (e.g., a boolean circuit), suitable for the evaluation process.
 - **PrepareInput**: This method converts the local input of the participant in a suitable way for being used during the evaluation.
 - **VerifyRepresentation**: Depending on the specific protocol, the participants may be required to exchange the new representation of the function. In this case, this method needs to implement a verification procedure that checks whether the representation sent by another participant represents the original function.
 - **FunctionEvaluation**: This method evaluates the value of the function (e.g., GMW).

Implementation notes: The implementation of this functionality may require the specification of an ad-hoc language that should be used to specify the function to be computed.

5.8 Timestamping

Description: Timestamping allows to order the bit-strings temporally. Let a user have obtained a valuable bit-string (some sort of a signature). Let it also be known to that user that sometime in the future, this bit-string has to be shown to be created earlier than some other bit-string (which may represent some sort of revocation). Then, this user should ask for a timestamp for this bit-string as soon as possible. The timestamp is associated with this bit-string and there is a (partial) temporal order on the set of timestamps. Later, when two bit-strings are compared, the one for which an earlier timestamp could be presented by the interested parties, is considered earlier.

There are three principal kinds of parties. Servers provide the timestamps to users that submit timestamping queries. The publication authorities store the digests of the servers' states from different moments in time, thereby preventing the loss or tampering with the timeline created by a server. To prevent the loss or subsequent tampering of the database of the timestamping server, a digest of the server's current state is regularly sent to several publication authorities.

Mechanics: To get a timestamp to a bit-string, the User submits this bit-string to a Timestamping Server and gets back a timestamp. From time to time, the Timestamping Server sends some information to Publication Authorities. We call these pieces of information *round digests*. The time intervals between them are called *rounds*. The Publication Authorities store the round digests together with some information about the order in which they arrived (this may be arrival times or sequence numbers). The Publication Authorities also disseminate this information. Given a timestamp and a round digest that presumably was created later than this timestamp, the Timestamping Server may be queried for a proof that there is a one-way relationship from the timestamp to the round digest. Two timestamps can be compared. This is normally done by checking, for which one we can find the earlier round digest that depends on that timestamp.

If the system contains several Timestamping Servers, then the temporal orders created by each of them are in principle independent, i.e., timestamps issued by different servers cannot be compared in general. This can be overcome if the servers regularly either ask the other servers to stamp their latest timestamp, or themselves issue timestamps to round digests of the other servers. The policies for such cross-stamping must be published and there also has to be a means to verify the adherence to policies (trial stamping) and to publish the verification results.

In principle, anyone can become a Timestamping Server. In practice, there should be motivational mechanisms which ensure that there will only be a small group of servers that cross-stamp frequently. Having just one Timestamping Server is also not desirable because availability is a critical issue.

API:

- User
 - `GetListOfServers`: Returns the list of known Timestamping Servers together with their quality of service (frequency and amount of cross-stamping) and price list.
 - `Timestamp`: Takes a Timestamping Server name and a bit-string and asks for a timestamp.
 - `IsTimestampFor`: Verifies whether the given timestamp is for the given bit-string.
 - `GetRoundDigest`: Asks the given Publication Authority for the digest of the given round of the given Timestamping Server.
 - `TimestampInRound`: Asks the given Timestamping Server for the proof that the given round digest depends on the given timestamp.
 - `CompareStamps`: Checks whether one timestamp is earlier than the other.
- Timestamping Server

- GiveTimestamp: Responds to a User’s timestamping query.
 - PublishDigest: Sends the digest of the last round to the Publication Authority.
 - ProveTimestampInRound: Responds to a User’s TimestampInRound-query.
- Publication Authority
 - SaveDigest: Adds the given round digest of the given Timestamping Server to its database. Assigns a sequence number to it.
 - GetDigest: Returns the given Timestamping Server’s round digest with the given sequence number.

Implementation notes: The proposed functionality could be in principle implemented either by linking the timestamps together, or by letting the Timestamping Server sign the pair of the to-be-timestamped bit-string and the current time. The second case has much worse issues with trust and does not survive the revocation of the key of the Timestamping Server, so the first case is preferable.

In the first case, the temporal order of timestamps is fixed by making each timestamp include a cryptographic hash of the previous one. In regular intervals, the Timestamping Server publishes the most recent timestamp it has created. After publication, the users may query for proofs (in the form of hash chains) showing that the timestamp they are holding is earlier than the published value.

Two timestamps from the same round may be considered incomparable. Depending on the implementation, it may be possible to compare two timestamps from the same round as well. To prevent the loss or subsequent tampering of the database of the Timestamping Server, a digest of the server’s current state is regularly sent to several Publication Authorities. These Publication Authorities are assumed not to form large colluding coalitions. Therefore, if the servers send the digests to many Publication Authorities, and the users query many of them, everyone can be assumed to have the same view on what the rounds’ digests are. To compare timestamps one needs the proofs that they belong to certain rounds. These proofs can be obtained from the same Timestamping Server that created these timestamps. A good implementation queries for these proofs as soon as possible, even before any comparison request has been made. This guards for the disappearance of the Timestamping Server at a later time.

Research has to be done on how to find the best trade-off between granularity, availability and implementation costs. In the ideal case, there should be incentives for the optimal amount of servers to emerge and to cross-stamp with each other in the optimal fashion. The optimal amount of servers is certainly more than one because server availability is a big issue in timestamping.

5.9 Summary

The next table summarizes the main functionalities and API methods described in this section.

Functionalities	Entities and API methods
Anonymous communication	RelayInformation, SendInformation, CloseChannel CreateAnonymousChannel, AnonymousReply
Certified information access	CertifiedDB: GetId, SendId, SendAnswer, DeletePublicInfo GeneratePublicInfo, CertifiedAnswer, SendPublicInfo User: GetPublicInfoId, GetPublicInfo, CertifiedQuery GetCertifiedAnswer, VerifyAnswer PublicInformationStorage: GetInfo, DeleteInfo, SendInfo GenerateId, SendId
Reputation management for content filtering	User: Introduce, Label, Object, TakeResponsibility RevealPath, FilterOut Central server: CallForResponsibility
Reputation management for resource allocation	GetReputation, UpdateReputation ShowReputation, Decide,
Secure broadcast (Byzantine agreement)	User: BroadcastMessage, ReceiveMessage
Secure function evaluation with limited trusted third party	TrustedFunctionRepository: SendFunction, SendTTPFunction TrustedThirdParty: ReceiveTTPFunction, TranslateTTPFunction FunctionEvaluation User: DefineIdentities, ReceiveFunction, IdentifyTTP TranslateFunction, VerifyRepresentation PrepareInput, FunctionEvaluation
Secure function evaluation	TrustedFunctionRepository: SendFunction User: DefineIdentities, ReceiveFunction, TranslateFunction PrepareInput, VerifyRepresentation, FunctionEvaluation
Timestamping	User: GetListOfServers, Timestamp, IsTimestampFor GetRoundDigest, TimestampInRound, CompareStamps Timestamping server: GiveTimestamp, ProveTimestampInRound PublishDigest PublicationAuthority: SaveDigest, GetDigest

6 Functionalities for wireless subnetworks

In this section we propose a series of functionalities and system calls that will extend the overlay computing platform in order to include wireless and mobile nodes. We aim at providing a software framework that will permit developers to deal with the new and critical aspects that networks of sensors and tiny devices bring into global computing, and to provide a coherent set of high level services, design rules and technical recommendations in order to be able to develop the envisioned applications of global sensor networks and overlay sensing machines.

In our approach we consider such global networks as P2P systems where human users, intelligent agents, and powerful computers interact with wireless sensor networks, ambient intelligence environments and smart spaces. In particular, we distinguish the global network into three sub-domains:

- The P2P network where peers are applications executed on a powerful computer device that has the ability to communicate with other peers via internet or other global networks (i.e., entities of the overlay computer).
- The Nano Peers (sensor devices) that form wireless sensor networks and communicate via the wireless (or optical) medium.
- The Gateway Peers that consist of nodes that have access to the wireless sensor network and allow interaction between the P2P peers and the nano peers.

The advantageous distributed characteristics of the system are achieved via a series of functions and mechanisms (services) which are activated by the system as a response to various kinds of events, processes, actions, applications that take place on it. We define a service as a unit of work done by a service provider (i.e., sensor device, gateway, peer) to achieve desired end results for a service consumer (i.e., gateway, another sensor device, a peer). Both provider and consumer are roles played by software agents. The results of a service are usually the change on the state of the software agent of the consumer but can also be a change of state for the provider or for both.

Rather than following a multi-tiered architecture, in which a number of tiers are operating in a specific hierarchical way and specific interfaces are used for communication across each layer, we choose to follow a less tightly coupled architecture in order to offer more flexibility to the system. *Interaction among services* is performed over all the levels of the system, in order to exchange necessary information for the successful accomplishment of each of them. Software agents (services) running on a peer are considered as independent modules that may interact with other services on the same peer and/or software agents executed by nearby peers.

The services related to monitoring and controlling the system's performance interact with almost all other computational, communication, distributed, and other performance active mechanisms in order to receive necessary information for the correct evaluation and

measurement of system's performance. Services for failures handling interact also with almost all other active computational, communication, distributed, and other performance services/processes in order to maintain the system's fault-tolerance. In order to achieve this, interfacing of the software agents is achieved via messages that are descriptive rather than instructive. We trade-off efficiency with extensibility by allowing the software agents to use a relatively rich vocabulary while interacting with each other.

Furthermore, due to the heterogeneity of the devices but also to the very nature of such global sensing systems, each peer may operate with a different set of software agents, i.e., provide a subset of the available services, and may provide different versions of a particular service, i.e., provide different quality and functionality. In this sense, we point out that:

- Each wireless device may operate different set of software agents, i.e., provide a subset of services
- Each wireless device may operate different versions of a particular service

At the lower levels, it is not necessary that all services are present in all the devices. In addition there are services that overlap in terms of offered functionalities but are however targeting different parts of the network, or different hardware/software technologies. Such an example is the Routing and Data Propagation & Query Dissemination low-level services. We can view Routing as a superset of Data Propagation that requires higher amount of memory and control messages.

The key point of our architecture are *Gateway peers* that appear on the global network as participants that have certain sensing and monitoring capabilities. The existence of the wireless sensor network is abstracted from the top layers and similarly the overlay computer is totally unknown to the nano peers. In this way we categorize services in two levels:

- Higher level services that are made available to the overlay computer by the gateway peers
- Lower level services used by the nano peers (within the wireless networks), that are transparent to the overlay computer (i.e., "internal services") and only available to the gateway peers

Sections 6.1 - 6.6 are devoted to high level functionalities (namely, buffering services, gateway for environmental monitoring and actuator control, SQL-like functionality, network statistics and management/control, virtual sensor networks and high-level network programming).

Sections 6.7 - 6.16 are devoted to low level functionalities (namely, routing, mobility prediction and control, support management, energy management / topology control, clustering / grouping, time synchronization, localization, code update, data propagation and query dissemination, sensing / monitoring and actuator control).

6.1 Buffering services

Description: Wireless networks often face network disconnections, due to several reasons (poor quality links, environmental noise, etc.). The nodes of the overlay network handle these intermittent network disconnections using some kind of buffering service. This buffering scheme applies to all data, i.e., both data and control messages. Also, this buffering applies to data that needs to be forwarded to another node of the overlay network as well. For example, assume that a node of the network wishes to get some information from another node in the network that collects information from a wireless sensor network. At some point after they start communicating, the connection between these two nodes is temporarily disrupted. The second node will hold its information and wait until the network connection is restored, then it will start transmitting again.

6.2 Gateway for environmental monitoring & actuator control

Description: Wireless sensor and actuator networks can be deployed for local monitoring and control of micromechanical devices. Scattered sensor devices have the capability to collect data. From a data storage point of view, one may think of a sensor network as a distributed database that collects physical measurements about the environment, indexes them, and then serves queries from users and other applications external to or from within the network. The database approach provides a separation between the logical view (naming, access, operation) of the data held by the sensor network and the actual implementation of these operations on the physical network. Diverse sensor network users and applications can focus on the logical structure of the queries they intend to pose and are relatively isolated from the details of physical storage and data networking on the volatile physical infrastructure of the network.

This high-level service provides access to such wireless networks and acts as a controlling center where data are routed to. The service is responsible for the gathering of all the readings coming from the sensor networks and the forwarding of the queries from the data tier to the devices. In other words, they act as gateways between the wireless sensor networks and the fixed part of the global computer.

Mechanics: The queries on sensor networks may aggregate data over a group of sensors or a time window, contain conditions restricting the set of sensors from contributing data, correlate data from different sensors and trigger activation of an actuator. A user interacting with this service will typically issue a sequence of queries in order to obtain all the necessary information. Distributed query execution is optimized for both resource usage and reaction time.

On the hardware level, a typical service provider consists of one wireless device connected to a desktop PC or laptop along with a network connection to the global computer. The wireless device attached to the gateway is necessary to communicate with the wireless network. Alternatively, an embedded platform can be used. Data retrieved from the

wireless network are stored in a relational database fashion, with tables organized in three categories:

1. Device-related tables that are used to store information regarding the technical characteristics of the network nodes. We aim to support heterogeneous networks, i.e., networks that consist of different kinds of devices (e.g., Telos, MicaZ motes etc.) which in turn may have different kinds of sensors and actuators attached to them (e.g., light, pressure, humidity, temperature etc.).
2. Query-related tables that keep track of active and historic queries. The service offers different type of queries that can be made by the user to the sensor and actuator network, and each query may refer to multiple devices and multiple sensors.
3. Sensor readings and actuator state tables that store the information received from the wireless network. Each record represents a reading coming from a specific device in the network concerning a single sensor or actuator.

Related functionalities: Data propagation & query dissemination, Sensing / monitoring & actuator control

6.3 SQL-like functionality

Description: From a data storage point of view, one may think of a sensor network as a distributed database that collects physical measurements about the environment, indexes them, and then serves queries from users and other applications external to or from within the network. This service is responsible for the organization and storage of data after sensing actions, how program interfaces to the sensor database look like and how queries are processed and served in an efficient manner. The advantage of the database approach is that it provides a separation between the logical view (naming, access, operation) of the data held by the sensor network and the actual implementation of these operations on the physical network. Diverse sensor network users and applications can focus on the logical structure of the queries they intend to pose and are relatively isolated from the details of physical storage and data networking on the volatile physical infrastructure of the network.

Mechanics: The queries on sensor networks may aggregate data over a group of sensors or a time window, contain conditions restricting the set of sensors from contributing data, correlate data from different sensors, trigger data collection or signal processing on sensor nodes and spawn subqueries as necessary. A user interacting with a sensor database will typically issue a sequence of queries in order to obtain all the information he/she wants. Distributed query execution is optimized for both resource usage and reaction time. Initially, the service will support five basic SQL operators: count, min, max, sum, average. These basic functions will be extended to support more sophisticated data analysis.

Related functionalities: Data propagation & query dissemination, Sensing / monitoring & actuator control

6.4 Network statistics & management / control

Description: Monitoring the condition of a wireless sensor network based on gross network metrics and also nano-peer local state is crucial for achieving good overall performance (i.e., at the application level) and also necessary to draw conclusions on the current state of the execution of a distributed algorithm. Also, we wish to allow the network administrator to adjust the operation of the network by modifying the parameters of the underlying protocols (e.g. cluster sizes, sleep schedules, encryption level etc.).

The service collects information related to the operation of the nano peers (e.g. information on the topology, available energy, neighborhood sizes, etc.), and allow the developer to program the network's performance to match the application's needs by modifying the operational parameters of the system. Such information can be used to further investigate the performance of the network and possibly draw conclusions on the current state of the execution of a distributed algorithm. The service allows to control the operation of the network by sending control messages to the wireless nodes. The control may be limited to the energy saving specification of the devices (e.g., the percentage of sleeping period) or extended enough to cover network operation parameters for the size of clusters, the rotation frequency of cluster heads, the time synchronization accuracy, etc. Depending on the level of details provided, the information can be used to execute an offline algorithm (e.g., resource management, scheduling, assignment) in order to fine-tune the performance of the network.

Mechanics: The collection of the status of the network is based on constructing global snapshots either periodically or on demand. Depending on the level of details provided, the information can be used to execute an offline algorithm (e.g. resource management, scheduling, assignment) in order to fine-tune the performance of the network. The control may be limited to the energy saving specification of the devices (e.g. the percentage of sleeping period) or extended enough to cover network operation parameters for the size of clusters, the rotation frequency of cluster heads, the time synchronization accuracy, etc..

Related functionalities: Support management, Mobility prediction & control, Energy management / topology control, Time synchronization, Clustering / grouping, Data propagation & query dissemination

6.5 Virtual sensor networks

Description: This service offers the ability to manage multiple wireless sensor networks, each with a different control center, under a common installation. This is done by introducing the notion of a virtual sensor network that hides the actual network topology and

allows the user to control the motes as if they were deployed under a single, unified, sensor network. This abstraction significantly reduces the overhead of administering multiple networks. Furthermore, the idea of a unified, virtual sensor network allows the integration of totally heterogeneous sensor networks, i.e., not only regarding different kind of sensors attached to the motes of the network, but also different kind of CPU architectures that communicate under different RF and optical devices.

Mechanics: For each sensor network, a unique ID is given to its respective control center. This sensor network ID helps to distinguish one mote from another, when they have the same mote ID but belong to different sensor networks, thus making it possible to manage different networks in a unified way.

6.6 High-level network programming

Related functionalities: Support management, Mobility prediction & control, Energy management / topology control, Time synchronization, Clustering / grouping, Data propagation & query dissemination

6.7 Routing

Description: This service provides the means to route information in wireless mobile ad hoc networks by simply modifying the Internet protocols to more complex multilevel hierarchical schemes. The goal is to support different wireless technologies enabling communication in cases where nodes have different transmission ranges (and bidirectional links are not available) but also operate in networks where all nodes have homogeneous resources and capabilities.

The design goals for ad hoc network routing protocols include minimal control overhead, minimal processing overhead, multihop routing capability, dynamic topology maintenance and loop prevention. Our goal is to make the service flexible enough to be able to shift its operation in order to emphasize one of these aspects or even accomplish different ones based on the actual needs of the network (or a part of it), such as security, energy-conservation, multipath etc.

Related functionalities: Support management, Mobility prediction & control, Clustering / grouping, Localization

6.8 Mobility prediction & control

Description: Mobility prediction can be used in order to improve the performance of the routing service in the case of mobile ad hoc networks but also the performance of the data propagation service for the case of sink mobility in wireless sensor networks (thus transforming it into data collection). By exploiting non-random behaviors for the mobility patterns that mobile nano peers and gateways exhibit, we can predict the future state

of network topology and perform route reconstruction proactively in a timely manner. Moreover, by using the predicted information on the network topology, we can eliminate transmissions of control packets otherwise needed to reconstruct the route and thus reduce overhead. Finally, in some cases, by forcing the gateways to follow certain motion patterns, data collection (and communication in general) can be improved in terms of energy and time efficiency.

Mechanics: The service uses mobility prediction to improve the performance of the communication services in the case of mobile nano peers and also allows the gateway peers to move in order to collect the sensed data instead of passively waiting to be propagated to them by the network.

Due to the limited memory capabilities of the nano peers and also the energy constrained wireless communication devices, mobility prediction is limited to local methods that rely on the exchange of short data within the neighborhood of each peer. Furthermore, the absence of any fixed infrastructure or any global positioning devices restricts the prediction methods to the input acquired from the accelerometer sensors available to the nano peers. Depending on the available hardware, if the service is allowed to control the motion of a small set of peers, forcing them to move acting as *helpers* for message delivery and data collection can help in cases of low densities of nano peers and obstacles blocking communication.

Related functionalities: Support management, Routing

6.9 Support management

Description: In many cases, the user mobility patterns and motion rate may have a significant impact on the performance of routing service. In particular, the effect of these parameters on routing protocols that try to maintain and dynamically update connectivity-related data structures (like the well-known AODV protocol and its variations/extensions) may become dramatic in the case of very high mobility rates and/or sparse networks, in the sense that the dynamic changes may be faster than the time available for dynamically updating the data structures and also in the sense that the connectivity of the network may be significantly affected and the network diameter may be significantly increased. In such cases, such routing protocols may become inefficient and/or erroneous, leading to low success rates and increased message delivery times. The support services provide an alternative approach for message delivery that takes advantage of the motion in the network, e.g., by forcing few hosts to move acting as helpers for message delivery, and in this way tolerate well (and in fact benefit from) high mobility rates and low densities. The service provides methods to organize the support (or helpers) in terms of size (i.e., increase/decrease support size by adding/removing mobile hosts), operation (i.e., how messages are synchronized among support nodes) and motion coordination (i.e., if

the nodes move as a swarm, or try to continuously maintain a particular formation).

Related functionalities: Routing, Mobility prediction & control

6.10 Energy management / topology control

Description: Probably the scarcest resource in a wireless sensor network are the energy reserves of the sensor devices, hence, an important operational requirement for energy efficient operation arises. Energy efficiency is mainly achieved by reducing the energy consumption of the devices using energy management protocols or by optimizing the network topology in order to reduce communication cost with topology control techniques. In the former case, the operation of each sensor device alternates between periods of very low power consumption (sleeping periods), where the device does not participate in the propagation process, and normal operation mode (awake period). In the latter case, topology control techniques adjust the transmission power of the sensor devices and/or identify and shutdown redundant nodes in order to reduce data propagation cost and eliminate redundant paths. In practice, the two approaches are tightly coupled and used in conjunction.

Mechanics: It is important that the system provides energy management/topology control functionality that can adjust to user defined criteria and application requirements (i.e., low latency, fine-grained monitoring) as well as sensor devices constraints and network characteristics (i.e., short transmission range, only local knowledge of the network topology, heterogeneous/dynamic networks with asymmetric links). Thus, when implementing such functionalities randomized and/or adaptive techniques based on local optimization criteria (i.e., the traffic rate, local network density, local energy reserves) are an invaluable tool. Also, self-organization techniques such as clustering and/or swarm intelligence can be used to achieve improved efficiency.

Related functionalities: Clustering / grouping, Data propagation & query dissemination, Sensing / monitoring & actuator control

6.11 Clustering / grouping

Description: The organization of devices in clusters/groups can provide the means to improve many operational aspects and implement higher level functionalities of a wireless network. Typically, network nodes are organized according to network topology or wireless connectivity criteria (i.e., sensors that can directly communicate to one another) or according to application criteria (e.g., spatial distribution, processing/sensing capabilities). These functionalities can be used for implementing routing hierarchies, coordinated energy conservation, efficient dissemination of sensing tasks, etc. Thus, the clustering/grouping functionality must be efficient, yet flexible enough in order to satisfy the desired operational requirements at a given time.

Mechanics: The service provides methods to form dynamic, ad hoc groups of nodes according to the requirements of a task and resource availability is at the center of many wireless network systems and applications. We will need networking protocols for nodes participating in a group to maintain communications despite mobility or node and link failure. Anticipated usage patterns and data rates may determine whether it should be infrastructure-based or infrastructure-less (e.g., using source-initiated ad hoc routing). Once a group is formed, we will need to develop protocols for nodes to join or leave the group, for groups to merge with others, or for a group to split into multiple subgroups.

Another problem is the maintenance of common state information about the nodes in a group (for example, the size of the bounding box of the group). As nodes move, such information should be maintained in a consistent and persistent manner and in such a way that can be efficiently located by all nodes in the group.

Related functionalities: Routing, Energy management / topology control, Localization, Data propagation & query dissemination, Sensing / monitoring & actuator control

6.12 Time synchronization

Description: Time synchronization is the problem of maintaining local clocks in each node of the network and keeping them all synchronized (absolutely or relatively) to a master clock. This is a critical issue in both mobile and wireless sensor networks since applications need some kind of collaboration between the network nodes, which in turn means some kind of acceptable time synchronization. Especially in wireless sensor networks, continuous network-wide time synchronization is necessary since each node may face software (e.g., resets) or hardware problems (e.g., an internal oscillator that has slightly different frequency than that of the other nodes). Wireless sensor networks with time-synchronized nodes can provide accurate reports, topology control and energy saving schemes, transmission schedules, etc.

Related functionality: Data propagation & query dissemination

6.13 Localization

Description: Another critical problem for mobile and wireless sensor networks is localization (positioning), i.e., the case where the nodes in the network are aware of their location. In some cases this problem is solved by using geo-location systems, like GPS. In the case where such systems are not available in every node of the network, the problem of localization translates into doing a distributed calculation of the position of each node in the network with the aid of specialized algorithms. Such algorithms usually make the assumption that there exist some special nodes that are already aware of their position. The localization service enables the generation of location-accurate reports from the network nodes, the use of geometric routing algorithms, topology control schemes, etc.

Also, in sensor networks, the knowledge of the location of each node allows for submitting location-specific queries to the network like “Watch for unusual sounds in area 51” or “All sensor with temperature over 100 degrees must report their location”.

Related functionalities: Routing, Energy management / topology control, Data propagation & query dissemination

6.14 Code update

Description: This service is responsible for the dynamic remote reprogramming of all the nodes in the network. This capability is very important, since the network can consist of a very large number of nodes that cannot be manually upgraded. Also, these nodes may have restricted resources, e.g., limited storage memory, which is inadequate for storing all the programs that nodes may need to execute throughout the network lifetime. This service could also provide the framework for building mobile agent programs, that propagate throughout the network. The code to be updated propagates through the network using a multihop protocol.

Related functionalities: Energy management / topology control, Data propagation & query dissemination

6.15 Data propagation & query dissemination

Description: This service is responsible for sending queries to the devices of the wireless network. We categorize all possible queries using two criteria;

1. the nodes they are targeted to and
2. the way information regarding the queries are reported to the gateway

The first category includes mote-specific and attribute-based queries, while the second category includes periodic sensing, event driven and query based queries. These two categories are overlapping and the actual sensor queries are combinations of these categories. The service supports the possibility of “canceling” a query that is currently active. The user is able to dispatch a special cancel-query message that contains one or more IDs referring to the queries that are to be canceled.

Mechanics: In the case of a node-based query, we are interested in targeting specific nodes by using their IDs. In this case, the service will send one packet per each node included in the query or may use multicasting techniques to improve the communication efficiency. An example of such a query is the following: “activate the ceiling fan for device 5”. On the other hand, attribute-based queries are not targeted to specific devices but instead to the whole network. In this case, only those devices of the network that match

the specific attribute will respond to the query. An example of an attribute-based query is “give me the temperature readings from all sensors that have light readings over 200”. Therefore, in order to reach all nodes, flooding techniques can be used.

Periodic-update queries pose another time constraint along with the start and stop time constraints, regarding the time of reporting to the gateway. They specify an interval time period between successive query reports. An example of a periodic update query is “give me the temperature readings every 125 msec”. Event-driven queries do not pose such specific time constraints, but instead use the notion of events in order to define the time to report back to the gateway. An example of such an event is “when door opens”.

Examples of possible queries to a sensor network include “give me the temperature and humidity readings from nodes where light reading is over 200 starting from 10:15 till 13:30” and “give me the light readings from node 4 when temperature reaches 30°C”. Thus, we have four types of possible queries, attribute-based periodic update queries, attribute-based event-driven queries, node-specific periodic update queries and node-specific event-driven queries. For the purposes of cancelling a query that is currently active, if the query that needs to be canceled is node-based, the message is sent directly to the device involved. On the other hand, if it is an attribute-based the message needs to flood the whole network.

Related functionalities: Gateway for environmental monitoring & actuator control, Energy management / topology control, Clustering / grouping, Localization

Implementation notes: The standard packet size in TinyOS is 36 bytes, leaving 29 bytes when using standard single hop communication and in most cases less when using a multihop protocol. The space left is sufficient for sending all types of queries however. Timestamps occupy 5 bytes, while mote IDs consist of 2 bytes and sensor type IDs of 1 byte. Events and attribute constraints can be expressed in the same way, occupying 4 bytes, 1 byte for the type of sensor ID, 1 byte for the relation used (e.g., equal, greater than, etc.) and 2 bytes for the value used in the constraint. There is also another type of messages used to stop specific queries, the cancel-query message, which consists only of the IDs of the queries to be stopped (more than one queries can be terminated at the same time). Because of the fact that a query can poll many sensors in the same device and one packet might not be sufficient for sending the readings from all the requested sensors, more than one packets can be sent by devices answering a query back to the gateway. The sequential messages will contain the readings that did not fit in the previous ones.

6.16 Sensing / monitoring & actuator control

Description: This service is located at the lowest level at the stack and is strongly coupled with the underlying embedded operating system. It is responsible for the control and operation of the sensors and actuators available to the wireless device. It provides

simple buffering functionality until the sensor readings are propagated within the wireless network and also provides algorithms for identifying false readings and conversion of hardware specific/raw data to desired units.

Another aspect of the service is distributed signal processing (also known as Collaborative Signal and Information Processing, CSIP), that is making use of processing power of the nodes to preprocess the measured data, obtaining a different, more compact form of representation of the observed data, or a reduction to relevant aspects.

Mechanics: Lightweight signal processing algorithms refer to methods that require relatively few floating-point computations and less memory storage than those that are floating-point intensive such as Fast Fourier Transform (FFT). The field of signal and image processing has developed many algorithms for image restoration and enhancement and for feature extraction and recognition. Some of these algorithms, such as those based on relaxation or wavelet computation, can be effectively decentralized and implemented in sensor networks.

Related functionalities: Energy management / topology control, Clustering / grouping

6.17 Summary

The next table summarizes the main functionalities described in this section.

Functionalities	
Buffering services	Mobility prediction & control
Gateway for environmental monitoring & actuator control	Data propagation & query dissemination
SQL-like functionality	Energy management / topology control
Network statistics & management / control	Clustering / grouping
Virtual sensor networks	Time synchronization
High-level network programming	Localization
Routing	Support management
Sensing / monitoring & actuator control	Code update

A JXTA at a glance

Introduction JXTA is an ongoing project started by Sun and comprises of a set of open generic peer-to-peer protocols that enable each network device to contact and cooperate as a peer. The motivation behind JXTA is to provide peer-to-peer application programmers with a common programming language that will facilitate peer communication. At the core resides a set of protocol specifications that are the strength of JXTA design, since a new peer-to-peer application does not need to implement communication primitives.

The supported protocols can be briefly described as follows:

- *Peer Discovery Protocol*: it is used in order to discover system resources which are being represented as advertisements. Resources can be peers, peer groups, pipes, services or another resource that has an advertisement. The protocol enables peers to search and discover advertisements that exist at other peers. It is the main discovery protocol for all peer groups that are defined by users.
- *Peer Resolver Protocol*: enables the peers to send and process general queries.
- *Pipe Binding Protocol*: it provides a mechanism to bind a virtual communication channel to a peer endpoint.
- *Peer Rendezvous Protocol*: it is responsible for publishing messages within a peer group. Although it is possible for different groups to have different ways to publish messages, this protocol allows peers to connect to a service (and, thus, to publish and receive messages) and controls publishing. Peer resolver and pipe binding protocols exploit this protocol in order to publish.
- *Peer Information Protocol*: it provides a set of messages concerning the status of a peer. This information can be used for commercial or internal development of JXTA applications, e.g., in a commercial application, this information can be used to detect the use of a peer service and charge the users accordingly.
- *Endpoint Routing Protocol*: it provides a set of messages that can be used to enable peer-to-peer message routing.

Architecture The architecture of JXTA is depicted in Figure A.

The JXTA core layer provides those primitives that are absolutely necessary for each peer-to-peer application. In an ideal situation, those primitives are accessed by all applications. The elements of the core layer are *peers*, *peer groups*, *advertisements*, *protocols*, *messages*, *pipes* and *identifiers*. This layer includes the six main protocols that are offered by JXTA. Though these protocols are implemented as services, we view them as part of the core layer since they are designed as core services.

The service layer provides those network services that are desirable but not necessarily part of each peer-to-peer application. These services provide functionalities which can be

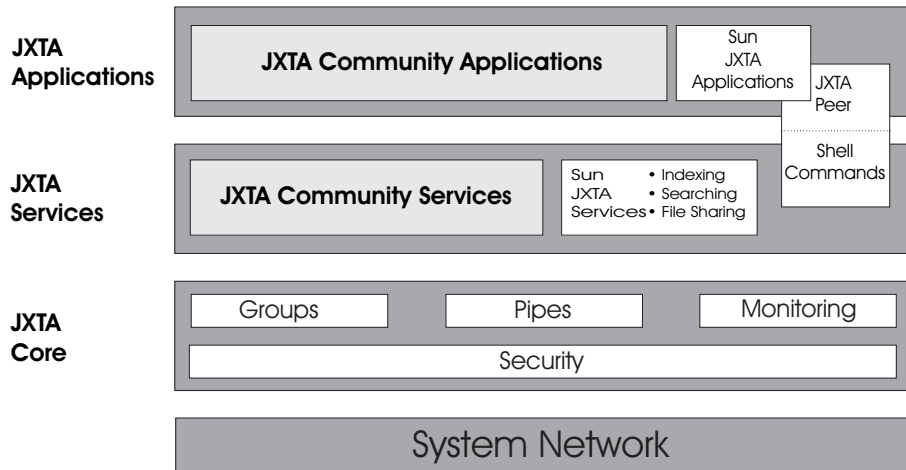


Figure 5: An architectural view of JXTA.

exploited by several applications like resource discovery, file sharing, peer authentication, etc.

Application layer is based on the functionalities provided by the service layer in order to support peer-to-peer applications. Since an application can include either only one service or several of them, it is not always feasible to distinguish an application from a service. Usually, the existence of a user interface is used as a separation criterion.

Main concepts *Peer groups* are a set of groups that have agreed upon a common set of services. The peers organize themselves into groups and can belong to more than one of them. When a peer enters the network, it is automatically assigned to group *Net Peer Group*. Groups enable peers to monitor other peers and exhibit a hierarchical parent-child structure, where each group has a unique parent group. Search queries are published within a group, while group advertisements are published to the parent-group as well as within the group.

Pipes are used by peers so that they can exchange messages. They are unidirectional, asynchronous message exchange mechanisms that are used for communication and data transfer, and can be used in order to transmit several types of messages, including binary code, data strings, java objects, etc. Pipes are virtual communication channels and can interconnect peers that have no physical connection. In this case, one or more additional peers are used for message retransmission between the pipe endpoints. Pipes support the following two ways of communication:

- Point-to-point pipes that connect exactly two pipe endpoints.
- Propagate pipes that connect a transmitting pipe to several receiving pipes. The entire publication procedure is conducted within the same group.

Messages are objects that are exchanged among peers and form the basic data transfer

means. There exist two possible message representations: binary and XML. JXTA J2SE platform uses the binary form while services use both of them.

Advertisements are used to represent all network resources, i.e., peers, peer groups, pipes, etc. They are meta-data structures represented by XML documents and are used by protocols to describe and publish the existence of a peer's resources. The way that peers discover resources is by searching for the corresponding advertisements. According to the resource they describe, advertisements can be classified to the following types:

- *Peer advertisements* are used to describe a peer. The advertisement contains information about the peer, e.g., his name, his identifier, possible endpoints and other attributes that services might want to exploit.
- *Peer group advertisements* are used to describe the resources that reside within a group and contain information like the group's name, identifier, description, etc.
- *Pipe advertisements* are used to describe a communication pipe.
- *Rendezvous advertisements* are used to describe a peer acting as rendezvous peer for a particular group.
- *Peer status advertisements* are used in order to maintain information about the current status of a peer

Rendezvous peers forward discovery requests so that other peers are able to discover resources. Each peer group has its own set of rendezvous peers and only those can identify search queries within the group. Peers send search and discover queries to rendezvous peers, which forward those queries that they cannot answer themselves to other rendezvous peers. This procedure ends when either a peer answers or the query expires. We can avoid sending the same query twice to a peer by maintaining a list of already visited peers.

Relay peers maintain information about how to route messages to other peers. A peer checks his local memory for routing information and in case no such information exists, he queries relay peers. Relay peers also forward information about peers that cannot be immediately addressed (e.g., firewalls, NATs).

References

- [1] J.D. Gradecki. *Mastering JXTA: Building Java Peer-to-Peer Applications*. Wiley, 2002.