Secure Space Computing with Exertions

Daniel Kerr and Michael Sobolewski
Texas Tech University, SORCER Research Group
sobol@cs.ttu.edu

***Abstract*–Exertion-oriented space computing is a valuable advance in distributed and parallel computing seeing as it abstracts out several major problems in distributed computing, such as load balancing and mutual exclusion. The main problem with space computing is that of security due to the fact that exertion spaces are inherently public and ad hoc, thus making it difficult to implement secure groups. The location independent group key interactive management framework presents a federated methodology and protocol for group management that is secure, scalable, and modifiable for the metacomputing exertion-oriented space computing environment. The framework does so through the use of a group establishment protocol, authorization and authentication services, high level cryptography, and persistent group information storage. The SORCER computing grid is used as a validation case for the framework and is presented in this paper.**

I. INTRODUCTION

Security in networking has been a serious issue since the dawn of networked systems. With the increasing trust in computers for the use of storing sensitive data (account numbers, social security numbers, criminal records, health history, etc) security has drastically increased in order to coordinate with the importance of the information being exchanged. Along with the growth in computer networking is the growth in distributed computing.

A rather new concept in distributed computing is the concept of space computing. Space computing helps to solve several of the significant problems with distributed computing, but still leaves room for improvement. The space computing environment starts by being completely public so that all services can access objects in the publicly shared space. The space broker is also oblivious to who is a member of the space environment.

Space computing, being based on the idea of the tuple space from Linda programming language, utilizes three major functions to manipulate tuples (or objects) from the tuple space (or object space). In Linda they are `in` – the removal of a tuple from the tuple space for reading, `out` – the writing of a tuple to the tuple space, and `rd` – the copying of a tuple from the tuple space for reading [[11]]. These three functions are mimicked in other implementations such as JavaSpaces [[3]] where the `in` function is called `take`, the `out` function called `write`, and the `rd` function called `read`.

The question at hand is whether or not a secure encrypted exertion oriented programming model that implements group creation and maintenance services can be implemented in a space computing environment.

The paper is organized as follows. Section II provides a brief description of the SORCER environment; Section III and IV describes basics of exertion-oriented programming; Section V describes three types of collaborations including space-based collaborations; Section VI presents the required cryptography and key agreement; Section VII describes a framework for the creation and management of groups within the exertion-oriented space computing environment, and Section 8 provides concluding remarks.

II. SORCER

SORCER (Service Oriented Computing EnviRonment) [[9]] is a federated service-to-service (S2S) metacomputing environment that treats service providers as network objects with well-defined semantics of a federated service object-oriented architecture. It is based on Jini semantics of services in the network and Jini programming model with explicit leases, distributed events, transactions, and discovery/join protocols [[6]]. While Jini focuses on service management in a networked environment, SORCER focuses on exertion-oriented programming and the execution environment for exertions. SORCER uses Jini discovery/join protocols to implement its *exertion-oriented architecture* (EOA) using *federated method invocation* [[8]], but hides all the low-level programming details of the Jini programming model.

In EOA, a service provider is an object that accepts remote messages from service requestors to execute collaboration. These messages are called service exertions and describe *service data*, *operations* and provider's *control strategy*. An *exertion task* (or simply a *task*) is an elementary service request, a kind of elementary remote instruction executed by a single service provider or a small-scale federation for the same service data. A composite exertion called an *exertion job* (or simply a *job*) is defined hierarchically in terms of tasks and other jobs, a kind of network procedure executed by a large-scale federation. The executing exertion is dynamically bound to all required and currently available service providers on the network. This collection of providers identified in runtime is called an *exertion federation*. The federation provides the implementation for the collaboration as specified by its exertion. When the federation is formed, each exertion's operation has its corresponding method (code) available on the network. Thus, the network *exerts* the collaboration with

the help of the dynamically formed service federation. In other words, we send the request onto the network implicitly, not to a particular service provider explicitly.

The overlay network of service providers is called the *service grid* and an exertion federation is in fact a *virtual metacomputer*. The metainstruction set of the metacomputer consists of all operations offered by all service providers in the grid. Thus, an exertion-oriented (EO) program is composed of *metainstructions* with its own *control strategy* and a *service context* representing the metaprogram data. The service context describes the data that tasks and jobs work on. Each service provider offers services to other service peers on the object-oriented overlay network. These services are exposed *indirectly* by operations in well-known public remote interfaces and considered to be elementary (tasks) or compound (jobs) activities in EOA. Indirectly means here, that you cannot invoke any operation defined in provider's interface directly. These operations can be specified in a requestor's exertion only, and the exertion can be passed on to any service provider via the top-level `Servicer` interface implemented by all service providers called *servicers*—service peers. Thus all service providers in EOA implement the `service(Exertion, Transaction):Exertion` operation of the `Servicer` interface. When the servicer accepts its received exertion, then the exertion's operations can be invoked by the servicer itself, if the requestor is authorized to do so. Servicers do not have mutual associations prior to the execution of an exertion; they come together dynamically (federate) for a collaboration as defined by its exertion. In EOA requestors do not have to lookup for any network provider at all, they can submit an exertion, onto the network by calling `Exertion.exert(Transaction) :Exertion` on the exertion. The `exert` operation will create a required federation that will run the collaboration as specified in the EO program and return the resulting exertion back to the exerting requestor. Since an exertion encapsulates everything needed (data, operations, and control strategy) for the collaboration, all results of the execution can be found in the returned exertion's service contexts.

Domain specific servicers within the federation, or task peers (*taskers*), execute task exertions. *Rendezvous* peers (jobbers and spacers) coordinate execution of job exertions. Providers of the `Tasker`, `Jobber`, and `Spacer` type are three of SORCER main infrastructure servicers, see

Fig. 1.

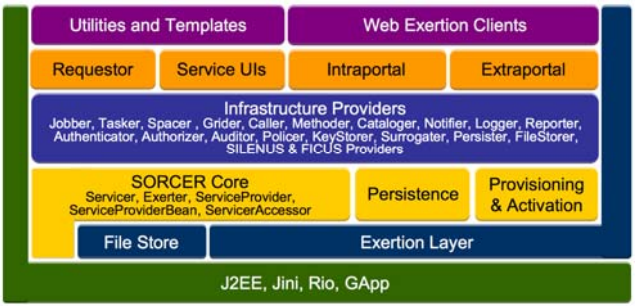


Fig. 1. The SORCER layered functional architecture.

In view of the P2P architecture defined by the `Servicer` interface, a job can be sent to any servicer. A peer that is not a `Jobber` type is responsible for forwarding the job to one of available *rendezvous* peers in the SORCER environment and returning results to the requestor.

Thus implicitly, any peer can handle any job or task. Once the exertion execution is complete, the federation dissolves and the providers disperse to seek other collaborations to join. Also, SORCER supports a traditional approach to grid computing similar to those found, for example in Condor [[10]]. Here, instead of exertions being executed by services providing business logic for invoked exertions, the business logic comes from the service requestor's executable codes that seek compute resources on the network.

Grid-based services in the SORCER environment include `Grider` services collaborating with `Jobber` and `Spacer` services for traditional grid job submission. `Caller` and `Methoder` services are used for task execution. `Callers` execute conventional programs via a system call as described in the service context of submitted task. `Methoders` can download required Java code (task method) from requestors to process any submitted context accordingly with the code downloaded. In either case, the business logic comes from requestors; it is a conventional executable code invoked by `Callers` with the standard `Caller`'s service context, or mobile Java code executed by `Methoders` with a matching service context provided by the requestor.

III. EXERTION-ORIENTED PROGRAMMING

Each programming language provides a specific computing abstraction. Procedural languages are abstractions of assembly languages. Object-oriented languages abstract entities in the problem domain that refer to "objects", communicating via message passing, as their representation in the corresponding solution domain. However, we cannot just take an object-oriented program developed without distribution in mind, and make it a distributed system, ignoring the unpredictable network behavior. The EO programming is a form of object-oriented

distributed programming that allows us to describe the distributed problem in terms of the intrinsic unpredictable network domain instead of in terms of distributed objects hiding the notion of the network domain that in reality cannot be hidden.

What intrinsic distributed abstractions are defined in SORCER? Well, service providers are "objects", but they are specific objects—they are network objects with a network state, network behavior, and network type(s). Service providers act also as network peers (servicers)*;* they are replicated and dynamically provisioned for reliability to compensate for network failures. Servicers can be found transparently in runtime by type(s) they implement. They can federate for an exertion submitted onto the network and participate in the collaboration outlined by the exertion. The exertion encapsulates service *data*, *operations*, and *control strategy* used by the collaboration. The component exertions may need to share context data of ancestor exertions, and the top-level exertion is complete only if all nested exertions are successful. Thus, a collaboration is a *process*, an exertion is the *specification* of collaboration, and a dynamic federation of peers is the *implementation* of a collaboration.

Let's first look at the EO approach to see how it works. Exertion-oriented programs consist of *exertion* objects called tasks and jobs. An exertion *task* corresponds to an individual network request to be executed on a service provider. An exertion *job* consists of a structured collection of tasks and other jobs. The data upon which to execute a task or job is called a s*ervice context*. Tasks are analogous to executing a single program or command on a computer, and the service context would be the input and output streams that the program or command uses. A job is analogous to a batch script that can contain various commands and calls to other scripts. Pipelining Unix commands allows us to perform complex activities without writing complex programs. As an example, consider a script `sort.sh` connecting simple processes in a pipeline as follows:

```
cat hello.txt | sort | uniq > bye.txt
```

The script is similar to an exertion job in that it consists of individual tasks that are organized in a particular fashion. Also, other scripts can call the script `sort.sh`. An exertion job can consist of tasks and other jobs, much like a script can contain calls to commands and other scripts.

Each of the individual commands, such as `cat`, `sort`, and `uniq`, would be analogous to a task. Each task works with a particular service context. The input context for the `cat` "task" would be the file `hello.txt`, and the "task" would return an output context consisting of the contents of `hello.txt`. This output context can then be used as the input context for another task, namely the `sort` command. Again the output context for `sort` could be used as the input context for the `uniq` task, which would in turn give an output service context in the form of `bye.txt`.

To further clarify what an exertion is, an exertion consists mainly of three parts: a set of *service signatures*, which is a description of operations in collaboration, the associated *service context* upon which to execute the exertion, and control strategy (default provided) that defines how signatures are applied in the collaboration. A *service signature* specifies at least the provider's interface that the service requestor would like to use and a selected operation to run within that interface. There are four types of signatures that can be used for an exertion: `PREPROCESS`, `PROCESS`, `POSTPROCESS`, and `APPEND`. An exertion must have one and only one `PROCESS` signature that specifies what the exertion should do and who works on it. An exertion can optionally have multiple `PREPROCESS`, `POSTPROCESS`, and `APPEND` signatures that are primarily used for formatting the data within the associated service context. A *service context* consists of several data nodes used for input, output, or both. A task may work with only a single service context, while a job may work with multiple service contexts since it can contain multiple tasks. The programmer can define a control strategy as needed for the underlying exertion by choosing relevant exertion types and configuring attributes of service signatures [[7]]. A reader interested in EO programming detail can review two simple EO programs for the `sort.sh` in [[7]].

If we use the Tenex C shell (`tcsh`), invoking the UNIX script is equivalent to: `"tcsh sort.sh"`, i.e., passing the script `sort.sh` on to `tcsh`. Similarly, to invoke the exertion `sortJob`, we call `"sortJob.exert()"`. Thus, the exertion is the program and the network shell at the same time, which might first come as a surprise, but close evaluation of this fact shows it to be consistent with the meaning of object-oriented distributed programming. Here, the virtual metacomputer is an ad hoc federation that does not exist when the exertion is created. Thus, the notion of the virtual metacomputer is encapsulated in the exertion (specification) that creates the required federation on-the-fly (implementation) to execute the collaboration (process).

IV. SERVICE MESSAGING AND EXERTIONS

In object-oriented terminology, a message is the single means of passing control to an object. If the object responds to the message, it has an operation and its implementation (method) for that message. Because object data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name (identifier) of the receiving object, the name of operation to be invoked, and its parameters. In the unreliable network of objects; the receiving object might not be present or can go away at any time. Thus, we should postpone receiving object

identification as late as possible. Grouping related messages per one request for the same data set makes a lot of sense due to network invocation latency and common errors in handling. These observations lead us to service-oriented messages called exertions. An exertion encapsulates multiple *service signatures* that define operations, a *service context* that defines data, and a *control strategy* that defines how signature operations flow in collaboration. Different types of control exertions (`IfExertion, ForExertion, and WhileExertion`) [[7]] can be used to define flow of control that can also be configured additionally with adequate signature attributes (*flow type* and *access type*—see Section V).

An exertion can be invoked by calling exertion's `exert` operation: `Exertion.exert(Transaction) :Exertion`, where a parameter of the `Transaction` type is required when the transactional semantics is needed for all participating nested exertions within the parent one, otherwise can be `null`. Thus, EO programming allows us to submit an exertion onto the network and to perform executions of exertion's signatures on various service providers indirectly, but where does the service-to-service communication come into play? How do these services communicate with one another if they are all different? Top-level communication between services, or the sending of service requests (exertions), is done through the use of the generic `Servicer` interface and the operation `service` that all SORCER services are required to provide—`Servicer.service(Exertion, Transaction)`. This top-level service operation takes an exertion as an argument and gives back an exertion as the return value. How this operation is used in the federated method invocation framework is described in detail in [8].

So why are exertions used rather than directly calling on a provider's method and passing service contexts? There are two basic answers to this. First, passing exertions helps to aid with the network-centric messaging. A service requestor can send an exertion out onto the network—`Exertion.exert()`—and any servicer can pick it up. The servicer can then look at the interface and `PROCESS` operation requested within the exertion, and if it doesn't implement the desired interface or provide the desired operation, it can continue forwarding it to another provider who can service it. Second, passing exertions helps with fault detection and recovery. Each exertion has its own completion state associated with it to specify if it has yet to run, has already completed, or has failed. Since full exertions are both passed and returned, the requestor can view the failed exertion composition to see what method was being called as well as what was used in the service context input nodes that may have caused the problem. Since exertions provide all the information needed to execute a task including its control strategy, a requestor would be able to pause a job between tasks, analyze it and make needed updates. To figure out where to resume a job, a rendezvous service would simply have to look at the task's completion states and resume the first one that wasn't completed yet.

V. PUSH AND PULL COLLABORATIONS

SORCER also extends exertion execution abilities through the use of a rendezvous service implementing the `Spacer` interface. The `Spacer` service can drop exertions into a shared object space, implemented using JavaSpaces [2], in which collaborating servicers can retrieve matching exertions, execute them, and return the resulting exertions back to the object space. When the attribute *access type* of a `PROCESS` signature is set to `PULL` then the associated exertion is passed onto a `Spacer`, otherwise (access type is `PUSH`) the exertion is passed directly on to the servicer specified by the signature. Another signature attribute—*flow type*, manages the flow of control (`SEQUENTIAL`, `PARALLEL`, or `CONCURRENT`) for all component exertions at the same level.

In Fig. 2, four use cases are presented to illustrate push vs. pull exertion processing with either `PUSH` or `PULL` access types. We assume here that an exertion is a job with two component exertions executed in parallel (sequence numbers with a and b), i.e., the job's signature flow type is `PARALLEL`. The job can be submitted directly to either `Jobber` (use cases: 1—access is `PUSH`, and 2—access is `PULL`) or `Spacer` (use cases: 3 —access is `PUSH`, and 4—access is `PULL`) depending on the interface defined in its `PROCES` signature. Thus, in cases 1 and 2 the signature's interface is `Jobber` and in cases 3 and 4 the signature's interface is `Spacer` as shown in Fig. 2. The exertion's `ServicerAccessor` delivers the right service proxy dynamically, either for a `Jobber` or `Spacer`. If the access type of the parent exertion is `PUSH`, then all the component exertions are directly passed on to servicers matching their `PROCESS` signatures (case 1 and 3), otherwise they are written into the exertion space by a `Spacer` (case 2 and 4). In the both cases 2 and 4, the component exertions are pulled from the exertion space by servicers matching their signatures as soon as they are available. Thus, `Spacers` provide efficient load balancing for processing the exertion space. The fastest available servicer gets an exertion from the space before other overloaded or slower servicers can do so. When an exertion consists of component jobs with different access and flow types, then we have a *hybrid* case when the collaboration potentially executes concurrently with multiple *pull* and *push* sub collaborations at the same time.

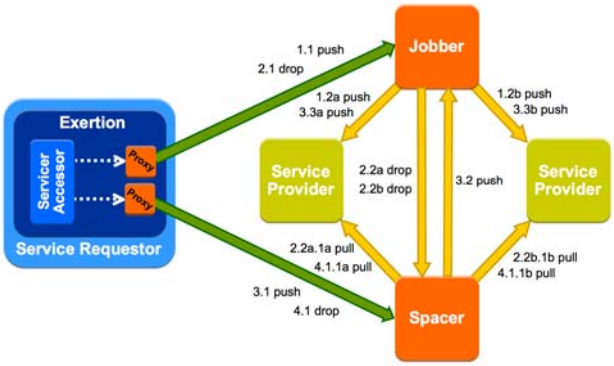


Fig. 2. Push vs. pull exertion processing

VI. CRYPTOGRAPHY AND KEY AGREEMENT

The major topic in cryptography necessary to understand the location independent group key interactive management (LOKI) framework is that of key exchange protocols. Key exchange protocol (also known as key agreement) is an agreement for how parties agree upon on a key. They require that each member must impact the key, that no previously exchanged information can be assumed, that it must prevent eavesdropping, and external parties should not be able calculate the key based on any publicly shared information.

When talking about SORCER `PULL` collaborations it is obvious that a flexible key exchange protocol is necessary in order to accommodate the creation of groups of unknown size. It is this requirement that prompted the creation of the Multi Diffie Hellman Key Exchange Protocol, which scales the standard two party Diffie Hellman Key Exchange Protocol into an N-ary key exchange protocol. It does so by linking the standard Java `KeyAgreement` object of each party member with all other party member's `KeyAgreement` in order to create a *Complementary Compound Key* (CCK) for each party member, which acts as a new public key for each specific member [[4]]. When a member's specific CCK is combined (`doPhase`) with its local Java standard `PrivateKey,` it will generate the group wide *Shared Secret Key* (SSK), which can then be used for both encryption and decryption, of shared objects. An example of this is shown in Fig. 3, and demonstrates a five party group.

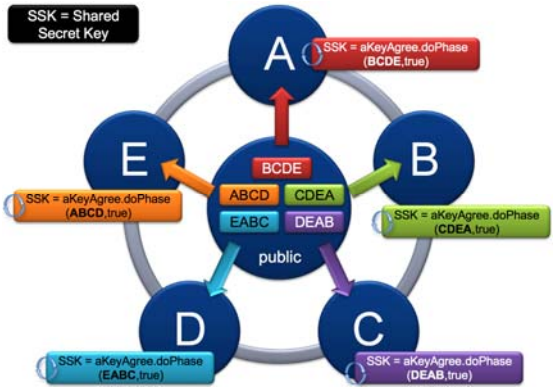


Fig. 3. Multi Diffie-Hellman Key Exchange Protocol

As you can see in Fig. 3, each party member's CCK consists of a chain of all other party members in the group. The main restriction on the Multi Diffie Hellman process is that it is more time consuming for groups of large size (greater than 1000). This is shown in Fig. 4.

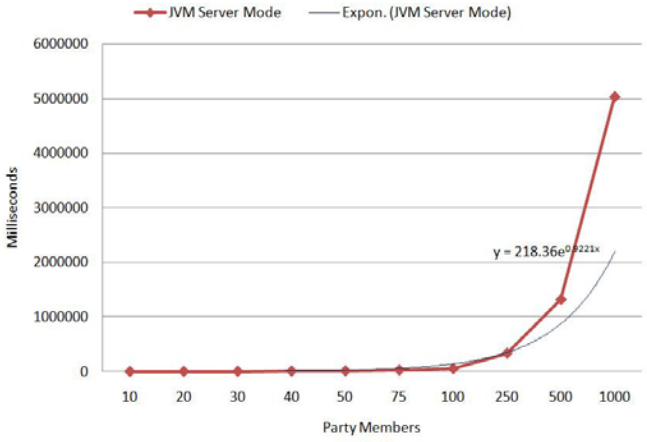


Fig. 4. Multi Diffie-Hellman Running Time Curve

Fig. 4, shows both the test data, in red, and the estimated line of best fit, in grey. The line of best fit shows that the Multi Diffie Hellman process is exponentially complex. Even though this is not ideal, this complexity is acceptable for the for the test case of LOKI implemented in the SORCER compute Grid (cGrid) application [[9]], due to the fact that groups are typically smaller than twenty five party members.

VII. LOKI FRAMEWORK

The LOKI framework establishes a protocol for group creation and management within `PULL` collaborations (see Section 5). There are several assumptions that are made about groups within the framework, the first being that there is a requestor, or a service desiring a group to be created. It is this requestor who will, by default be the administrator to the group that results from the creation request. The second

assumption is that the requestor knows what type of services or who it would like to invite to the group. From these two assumptions the LOKI framework handles the management of all subsequent services previously described in the exertion-oriented model.

To start we will look at what characteristics and attributes each service in LOKI has within the framework (see Fig. 5). Upon creation, each service will create a Java `KeyPair`, `KeyAgreement`, encryption `Cipher`, decryption `Cipher`, unique identifier, and status bits. This member information is utilized in order to perform LOKI secured read, write, and take operations to the exertion space. When a member performs a LOKI secured read, write, or take operation, it will first utilize discovery lookup to locate a `LokiGroupManagement` provider. The LokiGroupManagement provider is a service running within the space environment that is responsible for storing all group activity in a persisten data storage.

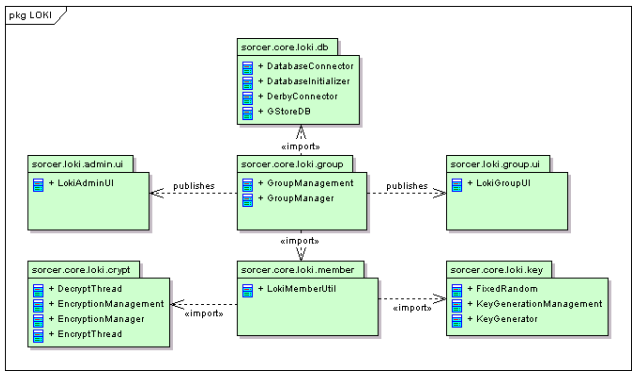


Fig. 5. LOKI package model

The use of persistent data storage allows for the framework to do continuity checking, ensuring that all data and group information is valid throughout group activity. If at anytime the group information is conflicting or faulty, the group is dissolved; all exertions are taken from the space and stored in the persistent data storage only to be accessed by the group administrator.

To understand the protocol of group creation in the simplest form it is best to start looking at the cGrid application without group implementation, then compare it to the implementation of LOKI groups in the LOKI cGrid (lcGrid) application.

A. SORCER Grid Collaborations

The `Grider` provider publishes a user interface which is used to enter the specifications for the job to execute. This information is stored into individual tasks, one for every block of data to be executed. These tasks are stored in a job and passed to the `Spacer` provider. The `Spacer` then breaks them down and dispatches them to the exertion space ("in" exertions in Fig. 6).

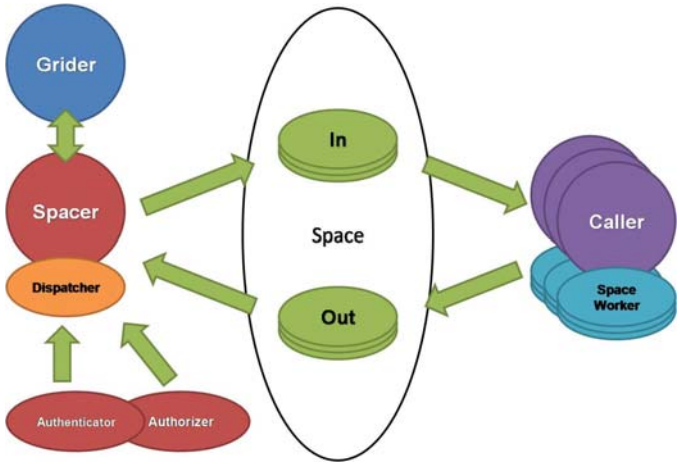


Fig. 6. cGrid Activity

The `Caller` service, upon startup, launches several `SpaceWorker`s that constantly listen to the exertion space for exertions with the `Caller` service type. At this point the `SpaceWorker`s start the collaboration between the `Spacer` and the `Caller`. The `SpaceWorker`s find the dropped exertion from the `Spacer` provider, take it from the space and call their own `exert` method for the retrieved matching exertion. Once execution is complete, the `Caller` service drops the results back to the space ("out" exertions in Fig. 6). The `Spacer`'s results collection thread, waits for these results and takes them from the exertion space. It is at this point that the `Spacer-Caller` collaboration is complete and the resulting information is processed. The results are then passed back to the `Grider` via the previously established proxy, where they are either presented to the user or persisted in the SORCER federated file system [[1]].

B. LOKI Enabled Collaborations

Execution of the cGrid application with LOKI starts in much the same way as it does without LOKI. The `Grider` provider publishes a user agent, which is used to enter the specifications for the job to execute. This information is stored into individual tasks, one for every block of data to be executed. These tasks are stored in a job and passed to the `Spacer` provider via `ServicerAccesor`. It is here where the first of two collaborations start between the `Spacer` and the `Caller (see` Fig. 7`)`. The `Spacer` takes the `PULL` collaboration attributes and sends out invitations or *Creator KeyPair* (CKP) exertions to the services required to execute the `PULL` collaboration, via the exertion space. Each invitation contains the `Spacer`'s public key, so that responses can be encrypted specifically for the `Spacer` to decrypt. The `Caller`'s `SpaceWorker` picks up this object, sees that it is an invitation, extracts the `Spacer`'s public key, encrypts its key pair with the `Spacer`'s public key, adds its own public key object, and drops this response

back to the exertion space (KP). The `Spacer` waits to receive a response for each invitation, then utilizes the `Authorizer` service [[2]] to validate that the responses are legitimate. It is here that the first collaboration is complete.

The `Spacer` service initiates the second collaboration in the LOKI protocol by combining the decrypted key pairs in order to calculate a CCK for each member of the newly created group. These CCKs are packaged into a CCK exertion, which is then dropped to the space. After this is complete, the `Spacer` begins to encrypt the initial job with the shared secret key, calculated by the `Spacer`'s private key and respective CCK, and drops the encrypted job exertion to the exertion space. The `Caller`'s `SpaceWorker` retrieves its respective CCK from the retrieved CCK exertion. The `Caller`'s `SpaceWorker` then retrieves the encrypted exertion from the exertion space and decrypts it with the generated shared key. The `SpaceWorker` passes it up to the `Caller`, where it's `exert` method is called, and results are generated. These results are then encrypted with the shared secret key, and dropped to the exertion space. The `Spacer` picks up the component results from the exertion space, decrypts them with the group shared key, and then encrypts the resulting top level exertion with the `Grider`'s public key. Once this is complete they are passed to the `Grider`, via the established proxy, where they are decrypted and either displayed to the user or persisted in the federated file system.

As described in the LOKI protocol there are two major collaborations between the `Spacer` and the `Caller` (see Fig. 7). The first is for the actual establishment of the group and exchange of member information. The second is for the processing of the initial job created by the `Grider`. The only difference between the LOKI execution collaboration and the standard execution collaboration (Section A) is the encrypting and decrypting of the component exertions in the job itself.

will walk through Fig. 7, step by step in order to visualize the execution process:

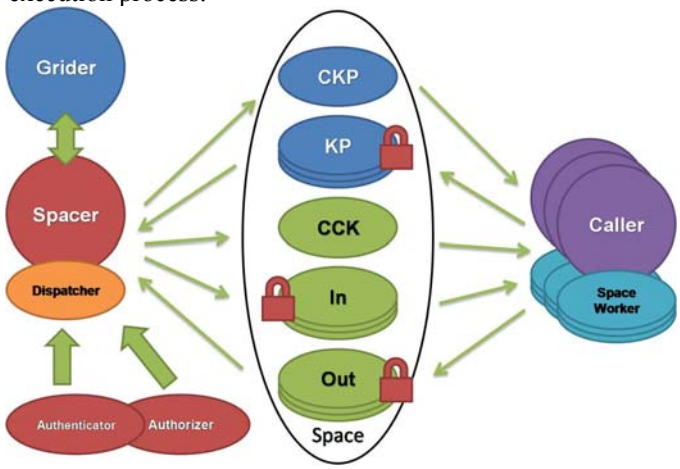


Fig. 7. LOKI cGrid collaboration

Step 1: Grider creates job and passes it to Spacer

Step 2: Spacer analyzes job, and writes CKP for every prospective group member

Step 3: Prospective member's SpaceWorker writes its KeyPair encrypted by group creator's PublicKey

Step 4: Spacer takes all KP exertions, computes CCKs, packs CCKs in CCK exertion and writes CCK exertion

Step 5: Spacer drops group wide encrypted job

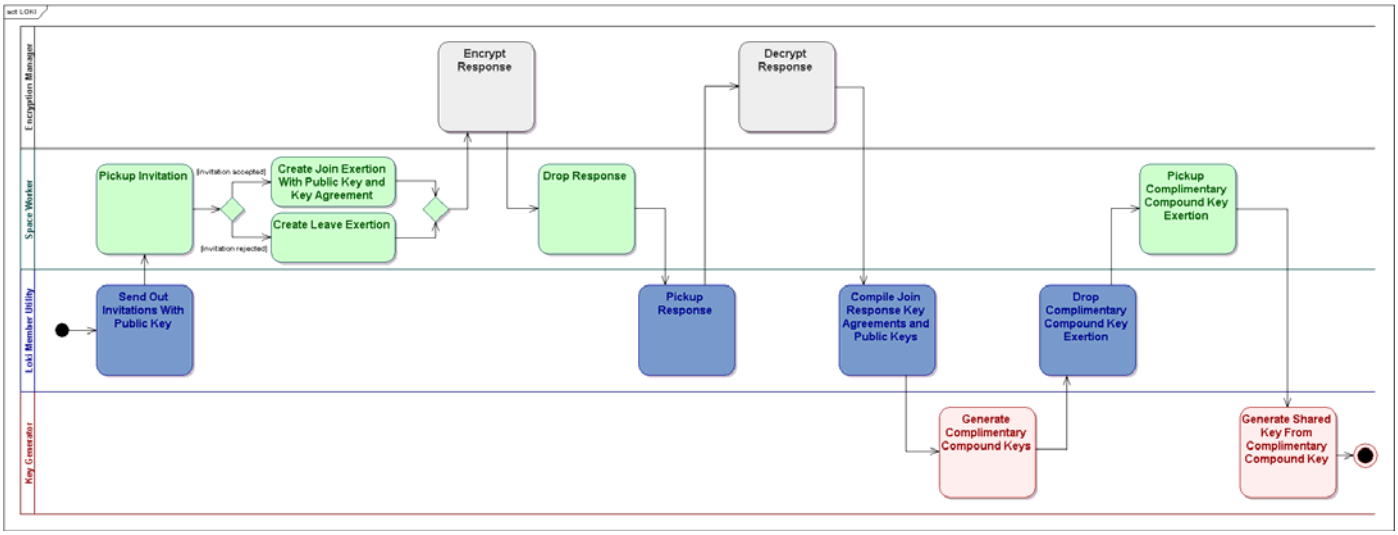
Step 6: Member's SpaceWorker reads CCK exertion

Step 7: Member's SpaceWorker takes encrypted job and decrypts with group wide shared key

Step 8: Member's Space Worker computes results, encrypts results and write to space.

Step 9: Spacer takes results and decrypt them with the group wide shared key

Step 10: Spacer passes the results to the Grider



C. *Security Flow Control*

Vital to the credibility of the LOKI framework, security needs to be maintained from start to finish. In order to verify that all links of the communication are secured in the LOKI implementation we must refer to Fig. 7.

The lcGrid job execution starts with `Grider` service, which then communicates with the `Spacer` service with the help of `ServicerAccesor`. The `Grider` to `Spacer` communication is secured with a two party Diffie Hellman key agreement. The Diffie Hellman agreement utilizes the `Grider`'s `KeyPair` and the `Spacer`'s `KeyPair` to individually calculate a common shared secret key. The shared secret key is then used to encrypt exertions exchanged between the two services.

The next link in the communication of job execution is that of the `Spacer` to `Caller` link, which passes over the exertion space. The `Spacer` to `Caller` communication is secured using the LOKI group key protocol (Fig. 8), and scalable key agreement. The LOKI protocol outlines the procedure for group creation, as well as the modified space interaction methods. These methods utilize the group's shared secret key to encrypt exertions passed between services over the exertion space.

The last possible gap in the job execution communication is in the concept of the invitation sent out by the group requestor. In order to ensure security, the validity of the response to the invitation needs to be authenticated. In order to do this within SORCER, the `Authenticator` service is utilized [[2]]. The `Authenticator` service guarantees that the service that has responded is the service that was invited.

VIII. CONCLUSION

The first objective was to secure the inherently public space computing environment. Through the implementation it has been shown that the LOKI framework successfully secures the lcGrid execution environment. The encryption and decryption of exertions with a securely created group wide shared secret key, before any exertion reaches the public space, ensures that exertions stay secure.

The second objective is to maintain security in the ad hoc space computing environment. The creation of the group key, accounts for the ad hoc nature of the system. The key is created and members who belong to the group can come and go, but will not be given access to the group if they establish new group characteristics (i.e. unique identifier, `KeyPair`, and `KeyAgreement`). The key agreement that is created in the framework is scalable to any number of party members, which satisfies the third objective—to abstract the complexity of the N-ary key policy.

As well the LOKI framework provides persistent data storage, management of group services, and is inherently easy to use. The solution also maintains several important architectural characteristics such as availability through extensive fault maintenance, modifiability through the use of interfaces, performance through the use of Java, security which has been described previously, testability through the modular nature of the framework, and usability through complexity abstraction.

Although this is the case, the solution grows incrementally more complex with increasingly large groups. The framework still holds but the time complexity for groups over 1000 members will grow faster than the benefits of the framework provide. It is for this reason that we must conclude that the framework is complete and practical, but for groups of large numbers of members (greater than 1000), alternate formations should be explored. Such alternatives may be the concept of sub groups, where sub group's shared key is the member key in a parent group. This would dissolve the issue of increasing time complexity and allow for groups of sizes much larger than 1000 party members. Other alternatives may include the CCK component calculation through the creation of specialized fast CCK calculation service within the LOKI framework.

In conclusion the Loki framework successfully secures the ad hoc space computing environments, with management of group services with little to no restrictions.

REFERENCES

[1] M. Berger, and M. Sobolewski, "Lessons Learned from the SILENUS Federated File System," *Complex Systems Concurrent Engineering*, Loureiro, G. and Curran, R. (Eds.), Springer Verlag, ISBN: 978-1-84628-975-0, pp. 431-440, 2007.

[2] M. Berger, and M. Sobolewski, "Group-based Security in a Federated File System," *2nd Annual Symposium on Information Assurance*, Albany NY, June 6-7, 2007, pp. 56-63, 2007.

[3] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces™ Principles, Patterns, and Practice*, Addison-Wesley, ISBN: 0-201-30955-6, 1999.

[4] J. Garms, D. Somerfield, *Java Security*, Wrox, 2001

[5] M.E. Hellman, "An Overview of Public Key Cryptography," *IEEE Communications Magazine* , pp. 42-49, 2002.

[6] "Jini architecture specification," Version 2.1. http://www.sun.com/software/jini/specs/jini1.2html/jini-title.html.

[7] M. Sobolewski, "Service-oriented Programming, SORCER Technical Report SL-TR-13," 2008. Available at: http://sorcer.cs.ttu.edu/publications/papers/2008/SL-TR-13.pdf.

[8] M. Sobolewski, "Federated Method Invocation with Exertions," *Proceedings of the 2007 IMCSIT Conference*, PTI Press, ISSN 1896-7094, pp. 765-778, 2007. http://sorcer.cs.ttu.edu/publications/papers/96.pdf

[9] M. Sobolewski, "SORCER: Computing and Metacomputing Intergrid," *10th International Conference on Enterprise Information Systems,* Barcelona, Spain, 2008. http://sorcer.cs.ttu.edu/publications/papers/2008/iceis-intergrid-08.pdf

[10] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," In Fran Berman, Anthony J.G. Hey, and Geoffrey Fox, editors, *Grid Computing: Making The Global Infrastructure a Reality*. John Wiley, 2003.

[11] G. Wells, Coordination Languages: Back to the Future with Linda, Rhodes University.

[12] D.R. Kerr, "Space Computing with Group Key Agreement - Location Independent Group Key Interactive Management (LOKI)," Master's Thesis. http://etd.lib.ttu.edu/theses/available/etd-03132008-163802