2007

Securing tuple space: secure ad hoc group communication using PKI

Kyle Morse

M.S. Project Report

Securing Tuple Space:
Secure Ad Hoc Group Communication Using PKI

Kyle R. Morse
Rochester Institute of Technology
Department of Computer Science
krm4686@cs.rit.edu

Committee:
Chair: Professor Alan Kaminsky
Reader: Professor Hans-Peter Bischof
Observer: Professor Stanislaw Radziszowski

Table of Contents

Overview

Secure group communication in an ad hoc network is a largely unexplored research area. While there exist group key exchange protocols, a recent M.S. project [8] showed that various forms of the Group Diffie-Hellman protocol become computationally infeasible quickly as group sizes increase. Moreover, currently available protocols were not designed to be implemented in an ad hoc network where nodes sporadically enter and leave the group. This project explores establishing secure group communication in an ad hoc network through public key infrastructure.

Public key infrastructure (PKI) provides a framework for establishing and authenticating secure communication between devices. A trusted certificate authority (CA) generates an identifying token, or certificate, for an authorized device. The certificate contains the device's public key and other identifying information and is digitally signed by the CA to prevent forging. This public key may be used to initiate secure communication with a device. In addition, any devices not possessing a valid certificate will be denied from communicating with the rest of the ad hoc group. PKI is often used in traditional client/server architectures – a browser establishing an SSL session with a secure web server for example. This project migrates the concepts of PKI to the ad hoc arena.

This project uses the tuple space distributed computing paradigm for all ad hoc group communication. A tuple space is a store of tuples, or lists of objects, from which consumers may read tuples matching filter criteria and to which producers may post new tuples [4]. An easily made physical analogy to this concept is that of an announcement board, where people may read flyers and post new ones. While several implementations of tuple spaces in Java exist, (IBM's TSpaces [5], for example) they employ a client/server architecture which is incompatible with the ad hoc network setting for this project. Professor Kaminsky's TupleBoard API [7] is an implementation of tuple space designed for developing ad hoc distributed applications in Java. This middleware library is the basis for all work done during this project.

While the main focus of the project is adding security to the TupleBoard library an equally important second step is the development of an application showcasing the newly added features. The market for digital music has exploded in recent years. Unfortunately, much of the media available from online music outlets (e.g. ITunes, Napster) is crippled by the plague of DRM (Digital Rights Management). Therefore I created an ad hoc music distribution network which does not infringe on fair use rights but also provides resistance to piracy through the encryption, authentication and authorization mechanisms developed during the project.

Related Work

In order to establish secure group communication the exchange of encryption keys is required. In the simplest group, consisting of only two users the standard Diffie-Hellman key exchange may be used. In this protocol Alice chooses values a, g and p and sends g, p and $g^a \bmod p$ to Bob. Bob chooses value b and sends $g^b \bmod p$ back to Alice. Both parties may now compute the shared key as $g^{ab} \bmod p$.

The original protocol has some shortcomings. It is vulnerable to man-in-the-middle attacks if authentication is not performed on the exchanges. In addition, it does not scale well when expanded from the simple one-to-one case described above to the many-to-many case found in the group setting. In [8] Kim explores the use of and improves two Group Diffie-Hellman schemes – Authenticated Group Diffie-Hellman (A-GDH) and Strong Authenticated Group Diffie-Hellman (SA-GDH). Through timing measurements Kim concludes that for small groups these protocols may suffice, but any system designed to support groups larger than 20 – 30 nodes will not be able to use them due to the rapid growth of key exchange time as nodes are added.

In addition to the initial key exchange keys should also be refreshed periodically. If forward and backward secrecy is of concern a refresh must also be performed each time the group membership changes. The sporadic nature of ad hoc group activity creates an environment in which keys must be refreshed quite often. It is therefore even more important that exchanges be performed as quickly as possible.

There also exist tree-based key management schemes which build on the concepts presented but also attempt to minimize the cost of key exchanges and refreshes by aggregating all refreshes in a specific time window into one or limiting the scope of the refresh to subgroups [9]. Examples of these protocols include Tree-based Group Diffie-Hellman (TGDH), Block-Free Tree-based Group Diffie-Hellman (BF-TGDH) and Distributed Scalable Secure Communication (DISEC).

In a PKI based system such as the one developed for this project group symmetric key exchange is trivial. Since the group private key is contained in each user certificate group symmetric keys may be posted encrypted with the group public key. In addition the authentication of group members and keys is provided via digital signatures on each message exchanged. Key refreshes are performed in the same manner as the initial key exchange, by posting a new encrypted symmetric key. Details of how these protocols are implemented appear in the next section, SecureTupleBoard API: Design & Implementation.

Secure TupleBoard API: Design & Implementation

When this project was started the TupleBoard API included only limited security. Package *edu.rit.crypto.hash* provides support for two hashing algorithms, SHA-256 and Double SHA-256, as well as an abstract *OneWayHash* base class for extensibility. Symmetric key encryption is supported by package *edu.rit.crypto.helix* which contains an implementation of the Helix stream cipher[3]. Tuples could be securely transferred over the network through the *FixedHelixSecurityContext* found in package *edu.rit.tb.security* which is backed by this cipher, however the symmetric key had to be agreed upon offline prior to the transfer.

The remainder of this section describes the security additions made to the library, namely secure pseudorandom number generation, public key cryptography and digital signatures.

Generating Randomness

A vital part of any cryptographic library is the ability to generate random data. Here, the term 'random' is used as a generalization. Truly random data is often hard to obtain and harder to evaluate its quality. Luckily it is easy to obtain pseudorandom data - seemingly random sequences of bytes deterministically generated from an initial seed. Java provides a pseudorandom number generator (PRNG) class *java.util.random* but this is not sufficient for use in cryptographic protocols.

An obvious application of a PRNG is creation of encryption keys. However a standard PRNG is often predictable, and if broken compromises the integrity of all keys it generates. To combat this problem we need a cryptographically strong PRNG, one that eliminates the predictability factor. While Java also provides a cryptographically strong PRNG in class *java.security.SecureRandom*, Professor Kaminsky asked me to create my own implementation for the TupleBoard API.

Luckily it is fairly straightforward to create a PRNG from a stream cipher such as Helix. Stream ciphers generally encrypt a plaintext by XORing it with a randomly generated key stream initialized with the encryption key. Here the key is analogous to the seed of a PRNG and the key stream represents the sequence of randomly generated bits. Class *edu.rit.crypto.helix.HelixRandom* provides a cryptographically strong PRNG based on the Helix cipher.

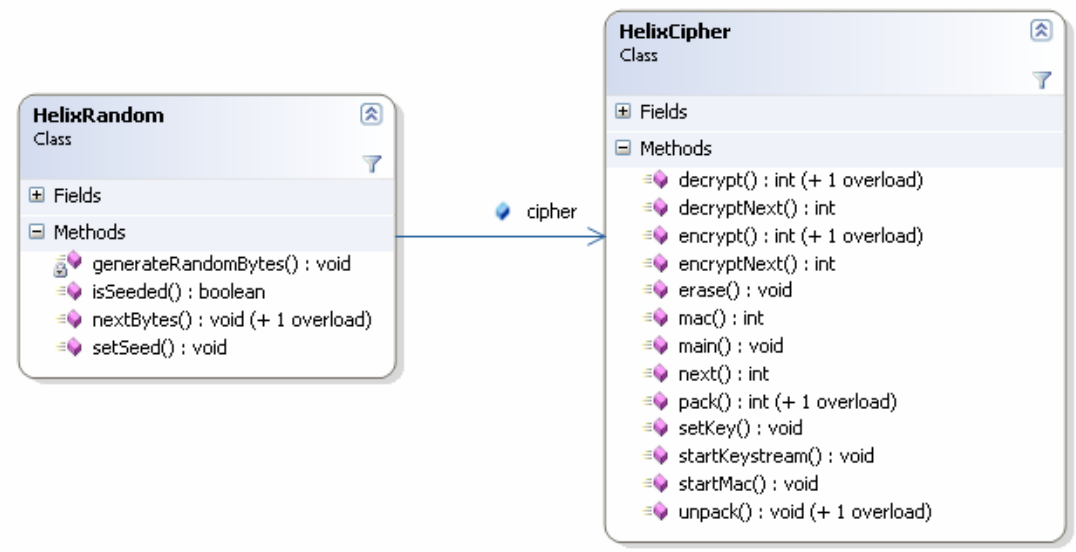


Fig. 1 – A cryptographically strong PRNG based on the Helix stream cipher

The concept for this algorithm was derived by modifying an algorithm found in Practical Cryptography [2] for creating a PRNG using a block cipher such as AES. When the PRNG is seeded it initializes a new instance of Helix using the specified seed. Random data is then retrieved through the *nextBytes()* method which simply encrypts the value 0 to obtain the XOR of the random key stream. The class is used throughout the TupleBoard library and is mentioned multiple times in this section. Guidelines for its use may be found in the Developer's Guide, found later in this document.

Public Key Algorithms

In order to have a public key infrastructure it is first necessary to have public key algorithms to back it. The primary difference of public key encryption algorithms from their symmetric key counterparts such as Helix is that the encryption and decryption keys are different. Also, the security of these algorithms is based on intractable computational problems such as factoring large numbers or solving discrete logarithms. Encrypting and decrypting data using a public key algorithm is much more computationally expensive than the same operations in a symmetric key system. Because of this the use of public key cryptography is generally limited in practical applications to situations where the amount of data to be encrypted is small. In the TupleBoard library it used for exchanging symmetric keys and digital signatures.

When designing this part of the API I tried to make it as general as possible to allow for additional algorithms to be added in the future.

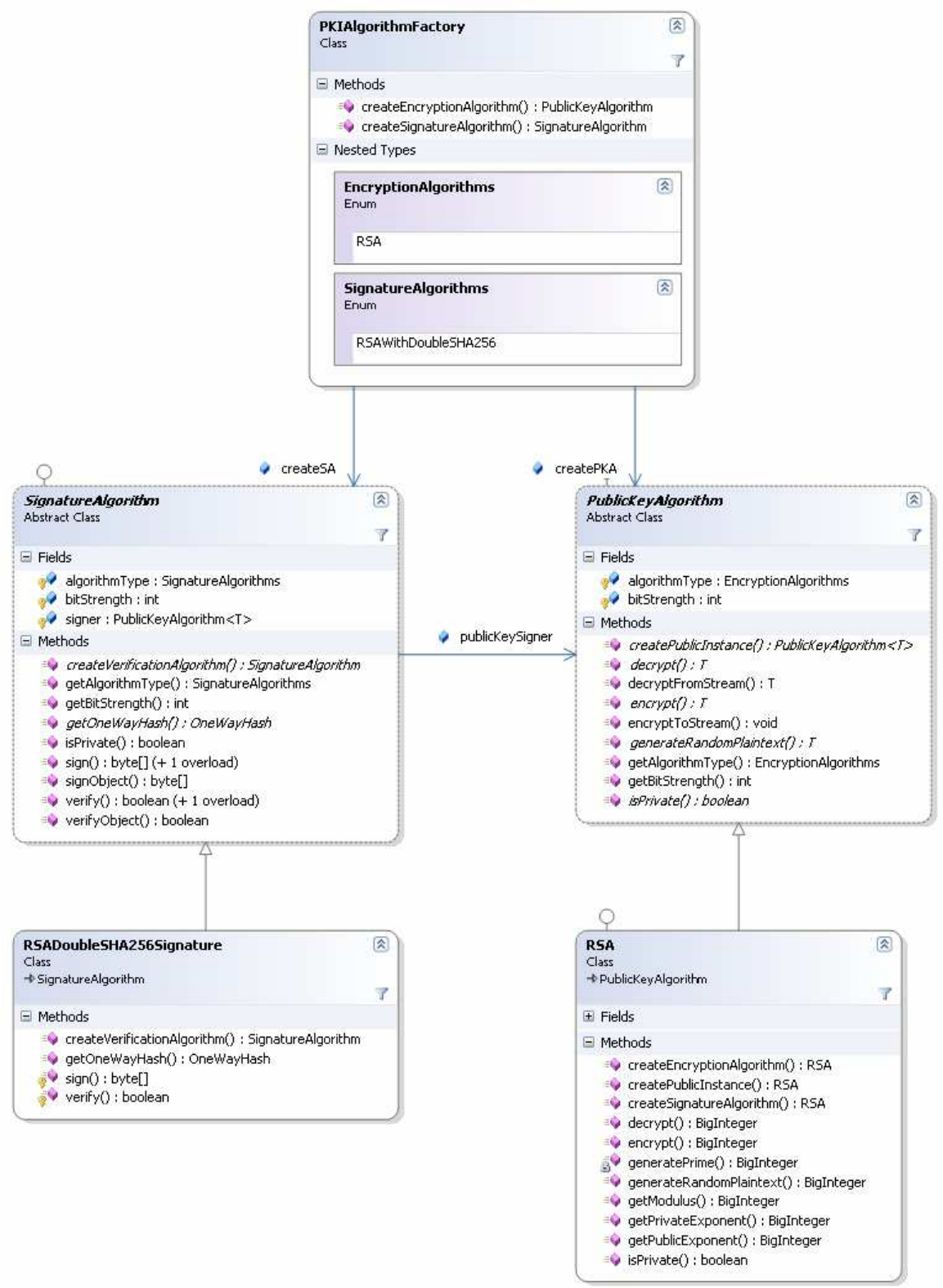


Fig. 2 – Public Key Algorithms

I created a factory class, *PKIAlgorithmFactory* to specify the available algorithms and simplify the creation of instances of the algorithms in the *edu.rit.crypto.pki* package. In addition I created abstract base classes for both public key encryption algorithms and digital signature algorithms. Class *PublicKeyAlgorithm* defines data members common to any algorithm such as the bit strength and provides an interface compatible with the new *PublicKeySecurityContext* defined later in this document. This allows any extending algorithm classes to be simply plugged in and used to securely post tuples to the tuple board. Similarly, class *SignatureAlgorithm* provides common signature algorithm fields and behaviors and allows for an extending class to compute and verify digital signatures using any combination of *OneWayHash* and *PublicKeyAlgorithm*.

RSA

With the interfaces defined the next step was to develop a concrete implementation of a public key encryption algorithm. RSA is probably the most well known algorithm in this category. It is also quite easy to implement in Java, thanks to the *java.math.BigInteger* class which already includes methods for performing modular exponentiation, primality testing and modular inverse calculation.

RSA is elegant in its simplicity. The basic parameters of the algorithm include *e*, the encryption exponent, *d*, the decryption exponent and *n*, the composite modulus. Encryption and decryption are both only a modular exponentiation operation. To encrypt: $c = m^e \pmod n$. To decrypt: $m = c^d \pmod n$, although this operation can be sped up using additional private parameters and Chinese remaindering.

The first step in implementing the algorithm is to generate the composite modulus, *n*. The algorithm I implemented for doing so is taken from Practical Cryptography[2]. The modulus is defined to be the product of two large primes, so the first step is to generate two large prime numbers, *p* and *q*. Each prime should be approximately *b/2* bits in length, where *b* is the desired length in bits of the binary representation of *n*.

The private method *generatePrime()* handles generation of these primes. First, an instance of the *HelixRandom* PRNG generates *b/8* random bytes which are then interpreted as a positive *BigInteger*. The resulting number is then verified to not be congruent to 1 modulo *e* and is tested for primality using the *BigInteger*'s *isProbablePrime()* method. If both of these conditions are true the number is returned, otherwise the method loops until a suitable number has been found.

Once both *p* and *q* have been generated the instance of RSA may be created. Public exponents are static in my implementation. Instances of RSA destined for use as encryption algorithms use $e = 3$, while those used as signature algorithms use $e = 5$. Public instances are initialized with only the public parameters: the encryption exponent *e* and the modulus $n = pq$. This instance is capable of performing encryption only.

Private instances are initialized with all the parameters: *e* and *n* as specified in the public instance as well as the two primes, *p* and *q*, the private exponent $d = e^{-1} \pmod{(p-1)*(q-1)}$ and $qInvP = q^{-1} \pmod p$. The last parameter is precomputed to increase efficiency of the

decryption computation. Rather than computing $m = c^d \pmod{n}$, we can compute $a = c^d \pmod{p}$ and $b = c^d \pmod{q}$ and then use Garner's Formula, $m = (((a-b)/qInvp) \bmod p)*q+b$ to recombine the modular parts and retrieve *m*. Timing measurements during implementation showed this CRT based implementation to be 2 to 3 times faster than the naïve approach.

For additional information on generating instances of RSA and its use for encrypting data please see the Developer's Guide found later in this document.

RSAWithDoubleSHA256

Once the implementation of RSA completed it was then possible to create a digital signature algorithm based upon it. Digital signatures offer two main benefits. First, they provide an authentication mechanism. If a message signature verifies correctly it confirms the identity of the sender of the message. Since only the sender would have known the private key used in signing, the signature must have been sent by the expected source. Second, they provide an integrity check on the contents of the message itself. Since the contents of the message are hashed during signing any tampering or loss of data during transmission will cause the verification to fail. If the message signature verifies correctly it confirms that the message itself is intact.

Class *RSADoubleSHA256Signature* represents a concrete instance of class *SignatureAlgorithm*. Like the name suggests the algorithm pairs an instance of RSA with an instance of *DoubleSHA256Hash*.

Class *SignatureAlgorithm* provides two methods for computing signatures. The first, *sign()*, takes any *OneWayHash* as input and generates a signature as a byte array based on the output of the hash function. The second, *signObject()* is a convenience method and takes any *Serializable* object. This overload simply serializes the object and writes its contents to the *OneWayHash* specified by the concrete class and then calls the *sign()* method to finish the computation.

Similarly, there are also two methods for verifying signatures. These function in exactly the same way as the signing methods. The first, *verify()* takes a *OneWayHash* and the signature as a byte array and returns the result of the verification as a boolean. The second, *verifyObject()*, is a convenience method and takes any *Serializable* object. This overload simply serializes the object and writes its contents to the *OneWayHash* specified by the concrete class and then calls the first *verify()* method to finish the computation.

The diagram on the following page details the steps involved in generating a signature for a message to be sent and verifying that signature when the message is received. Once again, the implementation for this algorithm comes from Practical Cryptography[2].

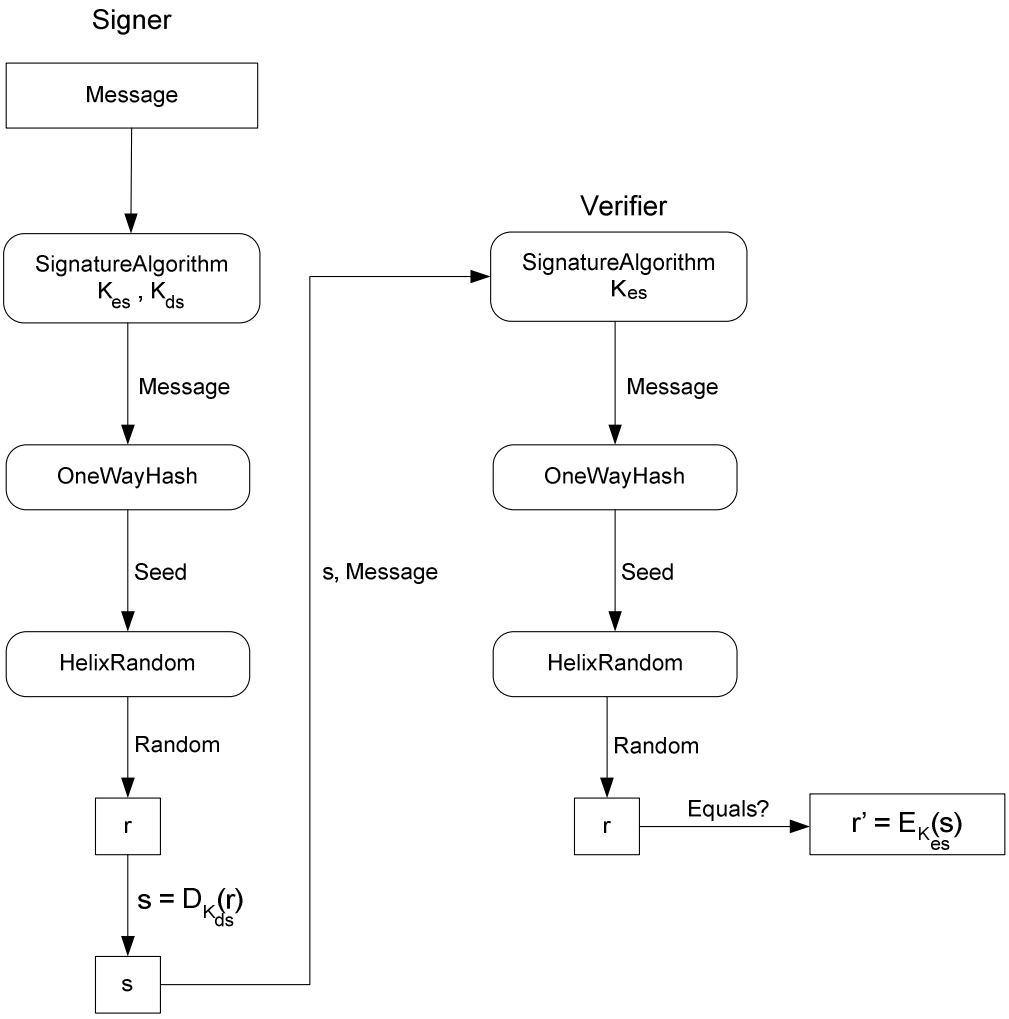


Fig. 3 – Computing & Verifying Signatures

To compute a signature first the sender hashes the message to be sent. The resulting hash is then used to seed a PRNG which is then used to generate a random plaintext element, *r* in the domain appropriate for the public key algorithm being used. The signature *s* is computed by decrypting *r* with the sender's private signing key, K_{ds}. The message and signature can then be sent to its intended recipient.

Once the message is received the signature is verified in much the same manner as it was originally computed. The message is hashed and used to seed a PRNG. From this the random plaintext element *r* is generated. Then the signature received, *s*, is encrypted using the signature verification key, K_{es} to yield *r'*. The signature is determined to be valid if *r'* == *r*.

Public Key Infrastructure

Public key infrastructure (PKI) provides a framework for establishing and authenticating secure communication between distributed parties. It is a trust based system wherein an empowered third party may create verifiable digital identities for users or devices. This third party is more commonly known as a certificate authority and the digital identities as certificates. Certificates are digitally signed so their content may be verified as authentic. They also bind public key information to a user or device, enabling secure communications to be initiated.

With the algorithmic parts of the public key system complete the next step was to define the infrastructure necessary for creating, distributing and using those algorithms for secure group communication. Both the certificate and certificate authority were designed to be flexible and accommodate any type of public key encryption and signature algorithms. Below is a high level design of the certificate authority and certificates added to the TupleBoard library in package *edu.rit.tb.security.cert*.

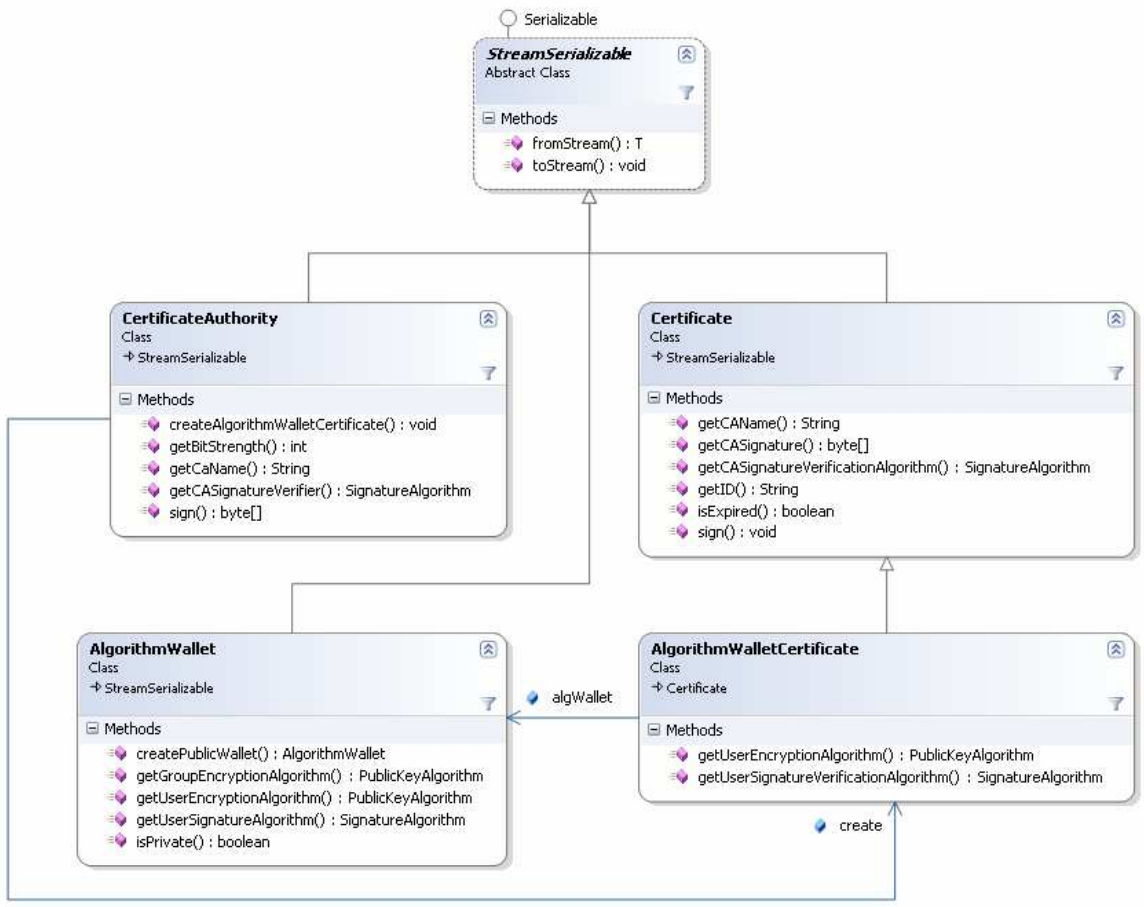


Fig. 4 – Public Key Infrastructure

The *Certificate* base class seen above contains the data fields and behavior common to any type of certificate based system. Each certificate contains the ID of the user or device for which it was generated and the name of the CA with generated it. Each certificate also has a start date and an end date, between which it is considered to be valid. The most important feature of the base certificate is its signature and the inclusion of a *SignatureAlgorithm* capable of verifying this signature. Through these the integrity of the certificate can be verified and the ID of its presenter confirmed authentic. A certificate is accepted as valid if the CA's verification algorithm matches the well known verification algorithm for the group, its signature verifies correctly using that algorithm, and the current date is after the start date but still before the end date.

While this simple certificate may be sufficient for very basic identification purposes it does not give the possessor the ability to receive encrypted messages or send signed messages in TB. Class *AlgorithmWalletCertificate* extends the base *Certificate* class and adds fields for a public key encryption algorithm and a signature algorithm for precisely this reason. Internally the certificate stores these algorithms in an *AlgorithmWallet*, which is simply a data holder class for public key algorithms. Instead of keys the wallet contains instances of the algorithms initialized with the necessary keys. The wallet may also be public or private. A private wallet contains public key algorithms capable of both encryption and decryption (and therefore capable of creating and verifying signatures). A public wallet may only encrypt data and verify signatures. In this manner the user may keep the private instance of the wallet but distribute the public instance in an *AlgorithmWalletCertificate* so other users may verify the user's signatures and also send encrypted messages capable of being decrypted only by that user.

With the certificates defined there now needs to be a certificate authority to generate them. Class *CertificateAuthority* provides this capability. The CA is created with a name, public key encryption algorithm and a signature algorithm. The CA's name serves as the group ID. The public key algorithm is included in all certificates created and allows group users to encrypt messages only other certificate possessing users have the capability of decrypting. The signature algorithm is used to digitally sign each certificate generated. The *createAlgorithmWalletCertificate()* method performs the creation of the certificate. Invoking the method results in the creation of two files – one containing the user's serialized private wallet and the other containing the user's serialized public certificate.

Finally, there needs to be a method of persistent storage for the certificate authority and certificates between application sessions. Serialization is a good candidate for this requirement, especially since certificates must be serializable to be sent over the network anyway. That is why all classes above derive from the common base class *edu.rit.io.StreamSerializable*. This class simply defines convenience methods *fromStream()* and *toStream()* for serializing and deserializing these objects to and from streams.

Please see the Developer's Guide and the User Manual found later in this document for more information regarding creating CA's and generating certificates.

Public Key Encryption in TB

With all the public key infrastructure pieces now in place a new *SecurityContext* had to be created to support transferring tuples using public key encryption. Class *PublicKeySecurityContext* enables the transfer of a tuple encrypted such that only a user possessing the corresponding private instance of a public key algorithm may decrypt its contents. As mentioned earlier, public key encryption is not used for bulk encryption operations, but is often used to transfer an encrypted symmetric key which the data will actually be encrypted with. Below is the design for the new *PublicKeySecurityContext*.

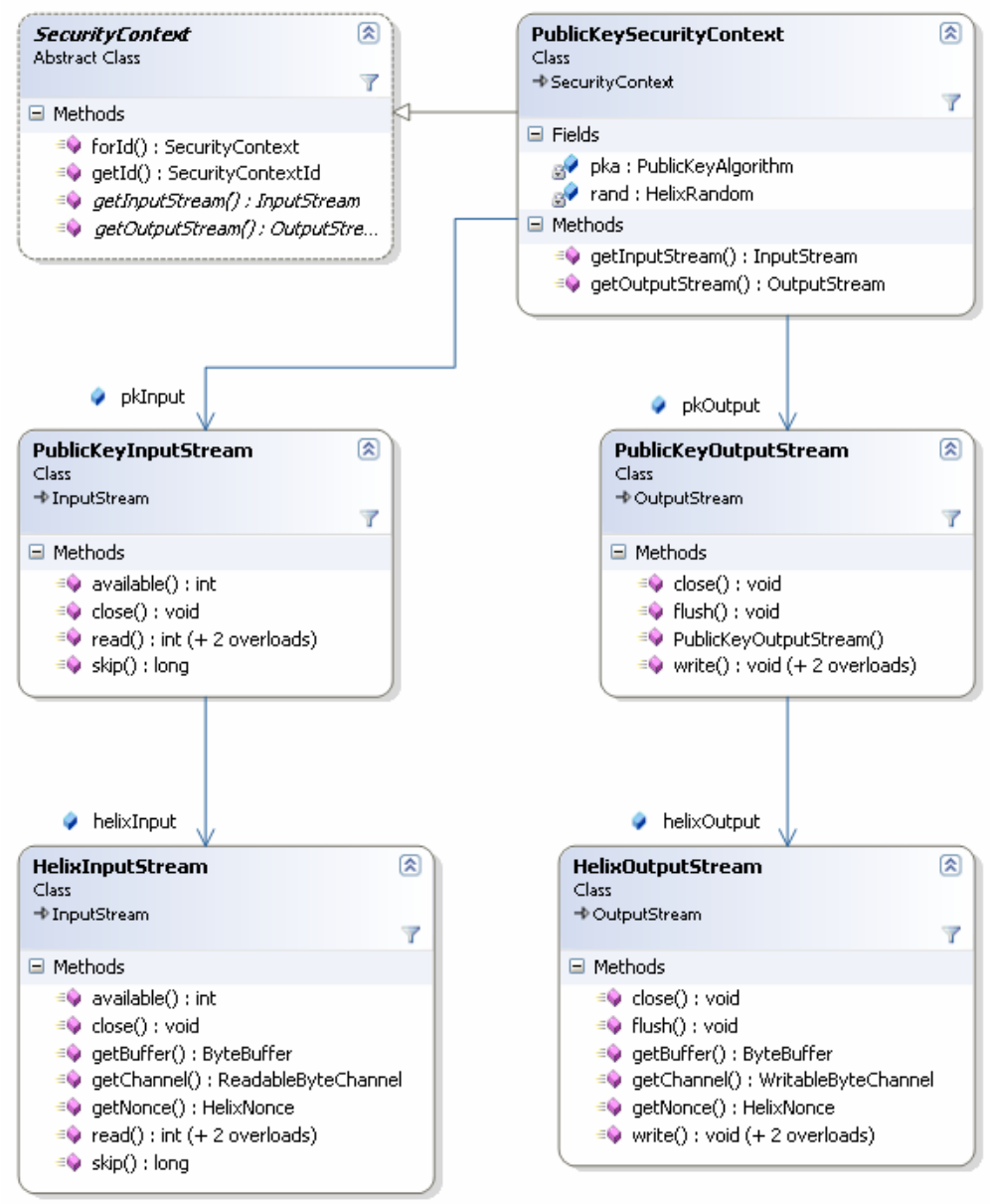


Fig. 5 – Public Key Security Context

A public key security context is initialized with a public key algorithm capable of encrypting data able to be decrypted only by the target user(s) and a PRNG to be used for generating a symmetric data transfer key. The *getInputStream()* and *getOutputStream()* methods simply return new instances of the *PublicKeyInputStream* and *PublicKeyOutputStream* classes pictured above. These classes handle the generation, encryption and decryption of the symmetric key and then delegate all further I/O to the *HelixInputStream* and *HelixOutputStream* classes. The implementation of these two classes is based largely on the protocol established between the two Helix streams.

The following diagram illustrates the steps performed when transferring a tuple posted to the tuple board using the *PublicKeySecurityContext*.

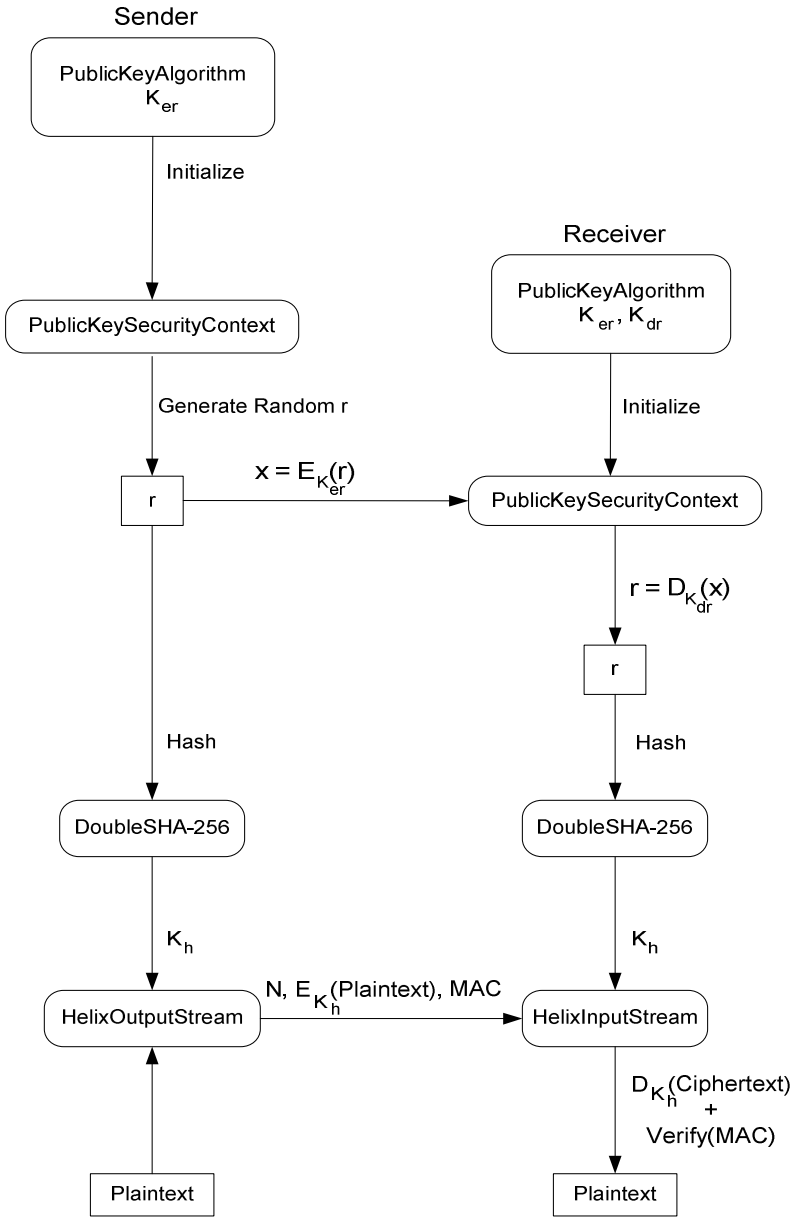


Fig. 6 – Public Key Encryption

First, the sender initializes a new instance of *PublicKeySecurityContext* with the public key algorithm of the target user and posts a tuple to the board using this context. When the tuple is read by the receiver a new *PublicKeyOutputStream* is created on the sender side. This object generates a random plaintext element *r* using the *generateRandomPlaintext()* method of the public key algorithm. It then computes *x* by encrypting *r* with the receiver's public key K_{er} and writes this value onto the stream.

Meanwhile on the receiving side a new instance of *PublicKeyInputStream* is created by the receivers corresponding *PublicKeySecurityContext* object. The input stream reads the value *x* and decrypts it using the private key K_{dr} to obtain *r*.

Now that both parties now have the value *r* the underlying symmetric key algorithm, Helix, may be initialized. The sender and receiver both hash the value *r* to obtain the symmetric key K_h. A new *HelixOutputStream* is initialized on the sending side with the symmetric key and a randomly generated nonce, *N*, which is transferred first. At the same time the receiving side initializes a new *HelixKeyInputStream* with the same key and reads the nonce value from the stream. From this point on all further I/O operations are delegated through these objects.

As an added bonus the Helix algorithm also computes a message authentication code, or MAC while encrypting a plaintext. When all serialized tuple data has been transferred the *HelixOutputStream* writes this code to the stream before closing it. The *HelixInputStream* on the other side can then read this code and compare it to its locally computed MAC to verify the authenticity and integrity of the received tuple.

Digital Signatures in TB

In order to support digital signatures in the TupleBoard library a new *SignedTuple* type was created. This class extends *Tuple* and so may be used in the same manner traditional tuples are used. *SignedTuple* adds fields for an *AlgorithmWalletCertificate* and a signature. The class may not be directly instantiated, as its constructor is protected. Subclasses should provide a static *create()* method in which the object is fully initialized and signed before being returned.

The signature is computed using the *SignatureAlgorithm* specified at creation time in the manner described previously. The message being signed is generally the serialized contents of the object itself. An example of this usage may be found in the Developer's Guide later in this document. Below is the overall design of the digital signature implementation in TB.

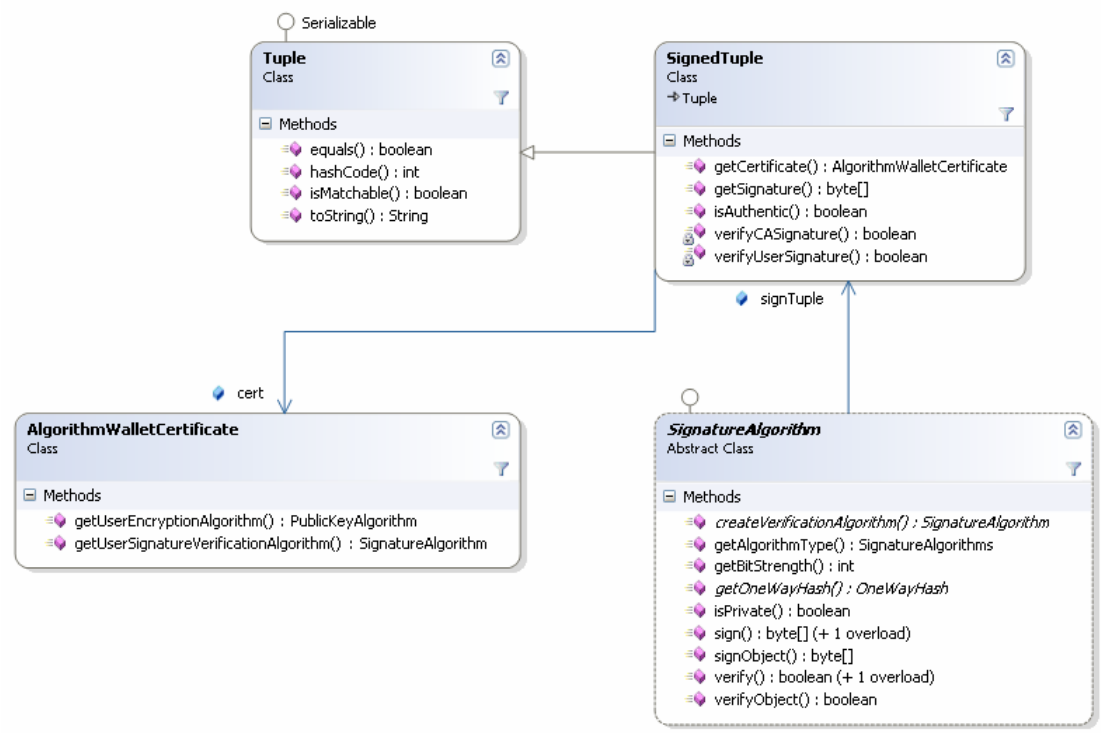


Fig. 7 – Digital Signatures in TB

When a *SignedTuple* is created and its signature initialized as described above it is ready to be posted to the tuple board. This is done no differently than a standard tuple. However, upon reading the signed tuple from the tuple board the receiver has the ability to verify its validity and authenticity. This operation is provided by the *isAuthentic()* method.

Three separate checks are performed by this method. First, the enclosed *AlgorithmWalletCertificate* is checked for expiration. If this test passes the CA's signature on that certificate is verified. If this check also succeeds then the signature on the tuple itself is verified using the signature verification algorithm found in the certificate. In this manner the authenticity of the tuple may be established.

Additional information regarding how to sign and verify tuples may be found in the Developer's Guide later in this document. Details on the specific instances of signed tuples used in this project are presented in the next section, Music Group Application: Design & Implementation.

MusicGroup Application: Design & Implementation

Once the security additions to TupleBoard API were complete the second phase of the project could begin. The goal of this phase was to create an ad hoc music distribution application utilizing the new secure tuple board. In this system role-based certificates would govern access to and permissions within music groups within the application. The system must be resistant to piracy by only permitting song downloads from authorized providers to authorized subscribers. In addition all communication must be encrypted to prevent unauthenticated users from gaining access to the secure group or any data transferred between users.

Song Metadata

The first step in developing the application was to define a set of metadata classes to represent music files.

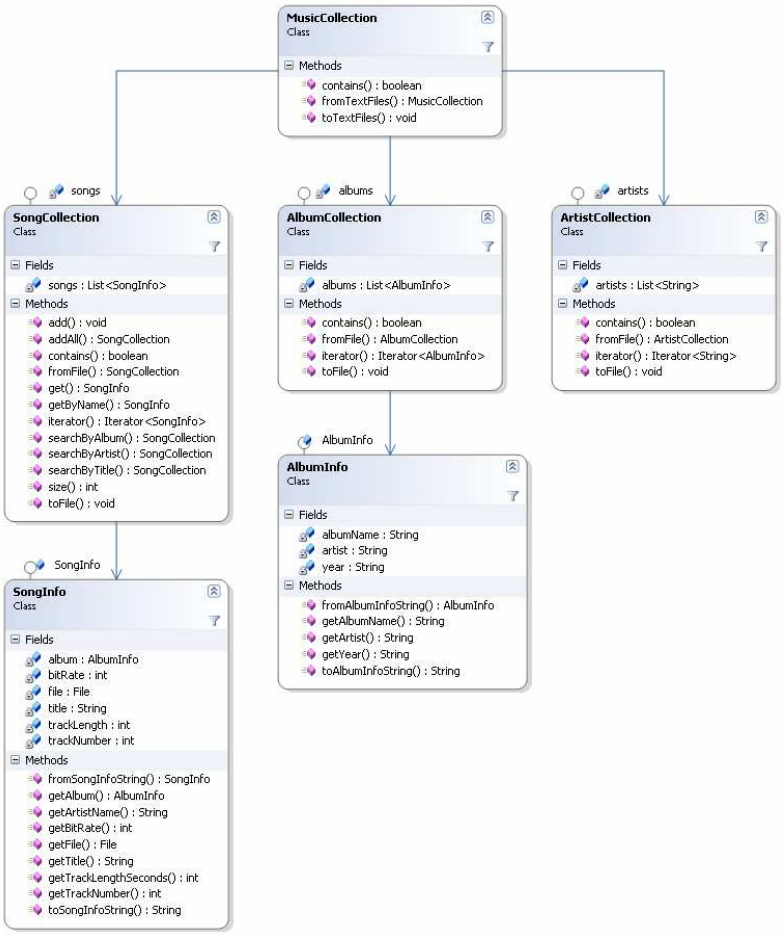


Fig. 8 – Metadata Classes

The metadata classes can be divided into two categories – entity classes and collection classes. Entity classes *SongInfo* and *AlbumInfo* contain useful information about songs and albums respectively. The remaining collection classes allow for entity objects to be easily stored, iterated over and queried. All classes are able to be serialized to disk or over the network.

The *SongInfo* class is the real workhorse of the metadata classes. This class contains a great deal of information about an mp3 file, including physical format data like bit rate and song length as well as data about the song itself, such as the album on which it appears (via an *AlbumInfo* object), the track number on that album and obviously the artist and song title. This class also has the ability to initialize its data members directly from an mp3 file by reading its ID3 tags and MPEG header data. This functionality is provided by the open source Java ID3 Tag Library[1].

The *SongCollection* class represents a collection of songs. This class is used widely in the application, from parsing local media libraries at startup to informing subscribers of available media from providers. Further details on how this is done appear later in this section.

The remaining collection classes were designed mainly for authorization duties. As mentioned previously song transfers in the system are only permitted between authorized users. A user is considered authorized to participate in a transfer if a *MusicCollection* object containing the requested song is associated with that user. *MusicCollection* objects are created and associated with users by a newly created music certificate authority discussed in detail a little later.

Roles and Authorizations

The application currently supports two roles, subscriber and provider. In addition to the general group activities supported such as sending and receiving public and private messages, subscribers are permitted to download songs. Providers are allowed to provide songs for download to subscribers.

In order for group activity to be regulated through roles and transfer permissions a new certificate authority and certificate had to be created to associate these permissions with users. Class *MusicAuthorizationCertificate* handles the binding of a role and a catalog of authorized music to a user. For a provider certificate the catalog specifies the songs the provider may distribute. Similarly, in a subscriber certificate it specifies the songs the subscriber may request for download. Class *MusicCertificateAuthority* enables generation of these new certificates. This certificate authority is analogous to a record label, permitting distribution of songs on a per user basis. The certificate is like a user license to provide or obtain songs. Below is a design diagram of these new classes.

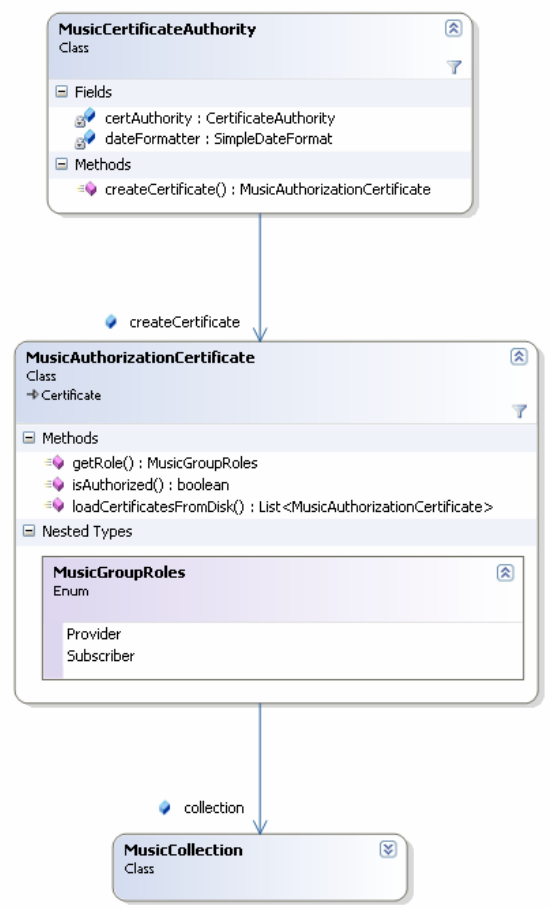


Fig. 9 – Music Authorization Framework

Each *MusicAuthorizatinCertificate* contains authorizations for one role and one *MusicCollection*. A group user may have one or more certificates specifying roles and the music authorizations associated with each. The class contains a static *loadCertificatesFromDisk()* method to provide easy support for deserializing these certificates for use in the application. The certificate also contains an *isAuthorized()* method for determining whether a specific song is authorized according to that certificate. Music authorizations are performed in a multi-tier fashion. A song is considered authorized if any of the following are true: the song's artist is found in the music collection's artists list; or the song's album is found in the music collection's albums list; or the song is found in the music collection's songs list.

Class *MusicCertificateAuthority* provides the ability to generate *MusicAuthorizationCertificate* objects. The certificate authority must be initialized with the instance of the group's certificate authority so that its certificate signatures may be verified by group members.

Certificates may be created using the *createCertificate()* method. More information regarding generating *MusicAuthorizationCertificates* may be found in the User's Manual later in this document.

Application Protocols

With the necessary roles and music authorizations defined tuples and protocols could be designed to enable collaboration between group users. Below is a diagram containing all application level tuples used in the system. Note that all tuples extend class *SignedTuple*, meaning all instances are digitally signed at creation before being posted and verified when read. Any tuples unable to be correctly verified are ignored.

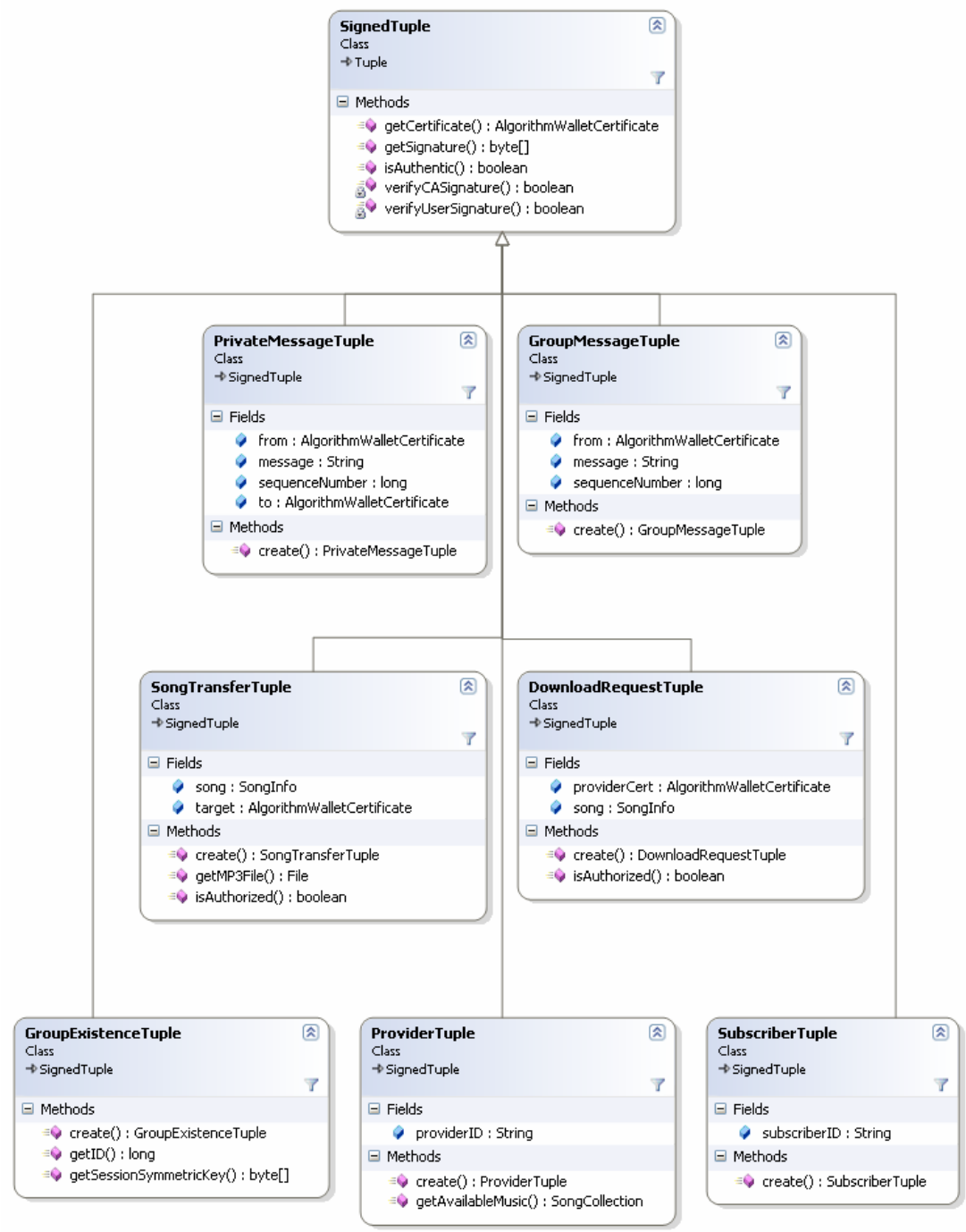


Fig. 10 – Application Protocol Tuples

Key Exchange Protocol

The first protocol necessary to facilitate group communication in the application is the establishment of a group symmetric key. Luckily this is trivial because of the public key infrastructure in place. Recall that each user has an *AlgorithmWalletCertificate* containing the public portions of its encryption and signature algorithms and a corresponding *AlgorithmWallet* containing the private portions. The wallet contains an instance of the group public key encryption algorithm. Using this algorithm it is possible to post encrypted tuples only other group members will be capable of decrypting. If this is the case why is it necessary to create a group symmetric key?

Technically it is not necessary. All communication could be done securely using the group public key algorithm however this is inefficient. Every time a tuple is read the associated *PublicKeySecurityContext* would have to decrypt the new random symmetric key and initialize a new Helix cipher. By creating a group symmetric key each user may initialize one instance of *FixedHelixSecurityContext* for all groupwide communication.

The establishment of a group symmetric key is done using the *GroupExistenceTuple*. When a user creates a new tuple board a read is performed for any existing tuples of this type. If the read succeeds the returned *GroupExistenceTuple* contains the current group symmetric key and the user may initialize an instance of *FixedHelixSecurityContext* using the read key. If no tuple is currently available, meaning the user is the first one to join, a new *GroupExistenceTuple* is created with a randomly generated key. This tuple is then posted to the board using a new *PublicKeySecurityContext* initialized with the group public key encryption algorithm and an ID of "GroupSymmetricContext" concatenated with the contents of the *keyID* field in the tuple. This additional ID field is included to prevent accidental reuse of an old security context identifier.

The protocol also allows for periodic refreshes of the group symmetric key. This feature is to prevent attacks which can arise when keys are used too many times. The frequency with which the key is refreshed is controlled by the system property "*tb.keyrefresh.interval*", specified in milliseconds. The default value is 900000, or 15 minutes.

A key refresh is performed the same way as the initial key generation. The original poster of the *GroupExistenceTuple* withdraws the old tuple, generates a new random key and reposts a new tuple. Meanwhile, all other group users are listening for tuples of this type and are notified of its posting and refresh their respective symmetric security contexts. Any tuples which were posted using the old context are also withdrawn at this time.

User Announcement Protocol

Once the symmetric key security context is initialized as described above the application may notify any other users currently participating in the group of its arrival. It may also gather information regarding the other users online and their roles. This is the essence of the user announcement protocol.

The two tuples involved in this protocol are the *ProviderTuple* and the *SubscriberTuple*. These tuples are posted to the board using the newly initialized group *FixedHelixSecurityContext* when a user authorized as a provider or subscriber begins participating in the group. Both tuples contain the user's *AlgorithmWalletCertificate*, enabling private communications with the user and the ability to verify signatures on tuples received from the user. The *ProviderTuple* also contains a *SongCollection* representing the catalog of songs the provider is authorized to provide and has available for download.

Existing group users are notified of the posting of these tuples through listeners added to the board during application initialization. The listeners also are notified when these tuples are withdrawn, signaling that a user has left the group. Through this protocol a current list of the users online and their roles may be maintained.

Secure Messaging Protocol

The application supports the sending and receiving of encrypted messages between users in the group through the secure messaging protocol. Two types of messages are supported. The first is a group message. This type of message is able to be read by all users currently participating in the group. The second type is a private message. This message is only able to be read by the intended recipient.

The *GroupMessageTuple* allows for messages to be sent to the entire group. It includes matchable fields for the originator of the message, the message itself, and the sequence number associated with the message. These tuples are posted using the group *FixedHelixSecurityContext* so all other group members may decrypt their contents and successfully read the message. Group users are notified of the posting of these tuples through a listener on the tuple board.

The *PrivateMessageTuple* is very similar to the *GroupMessageTuple* but allows for messages to be sent to a single user. In addition to the matchable fields of the *GroupMessageTuple* it also provides a field for the intended recipient. These tuples are posted using a *PublicKeySecurityContext* initialized with the public key encryption algorithm of the recipient. Since only this user has the private portion of the algorithm capable of decrypting the message it is unable to be read by anyone but the recipient. Group users are also notified of the posting of these tuples through a listener on the tuple board.

Since group messages are posted using the group symmetric key context, when a key refresh occurs all previously posted group messages are withdrawn. Private messages persist until the target user is found to be offline as described in the user announcement protocol above.

More information on sending and receiving secure group messages may be found in the User's Manual section later in this document.

Song Transfer Protocol

The song transfer protocol is probably the most important protocol in the application since its purpose is to enable the secure and authorized distribution of music between group members. It is also the most involved. While all previously defined tuple protocols involve only a single step this protocol consists of a request and a response.

A subscriber initiates the protocol by posting a *DownloadRequestTuple*. This tuple is initialized with the song wishing to be downloaded, a music authorization certificate which authorizes the download and the target provider that should field the request. Much like the private message protocol described in the previous section this tuple should be posted to the tuple board using a *PublicKeySecurityContext* initialized with the target provider's public key encryption algorithm. While this is not absolutely necessary since no truly sensitive data is contained in the tuple it does protect the privacy of the subscriber. No user but the target provider will be able to discover what song was requested or the collection of songs the subscriber is authorized to download.

The target provider is notified of the posting of any *DownloadRequestTuples* matching its *providerID* through a listener added to the tuple board during application initialization. When a new tuple is received it must be verified that the requesting subscriber is actually authorized to download the requested song. The tuple contains an *isAuthorized()* method which delegates to the provider authorization certificate to verify its authorization status as described previously in this section. The provider must also verify that it actually has the requested song and is authorized to provide it. If any of these conditions are not met the request is ignored.

Otherwise the provider creates a new *SongTransferTuple*. This tuple is initialized with the same *SongInfo* object specified in the request as well as the target and the *java.io.File* object containing the requested song in the provider's file system. In order to actually transfer the contents of the song file over the network some custom serialization must be done. This tuple is based directly on the implementation of the *FileTuple* found in the *edu.rit.tb.tuples* package.

The tuple specifies custom *writeObject()* and *readObject()* methods which are automatically invoked when the object is serialized and deserialized. In addition to the serializable fields of the object the contents of the song file are also written to the object output stream during serialization. Similarly, the contents of the song file are read from the object input stream when the tuple is deserialized. Since the signature on the tuple is computed by first serializing it the signature includes the file contents. Thus when the tuple is received both its authenticity and integrity of the file transfer may be verified. This tuple is posted to the tuple board by the provider using a *PublicKeySecurityContext* initialized with the subscriber's public key algorithm so no other users may decrypt and obtain the transferred file data.

When the subscriber reads this response tuple the song file's contents are written to a temporary file. This file is then copied to the location specified by the subscriber when the download request was first made. For more on how to download tuples using the application see the User's Manual section located later in this document.

Application Architecture

With the tuple protocols designed and implemented the remainder of the application could be developed. The application is written in multiple tiers in order to minimize coupling and increase reuse of developed components. The following diagram illustrates the manner in which the application is organized.

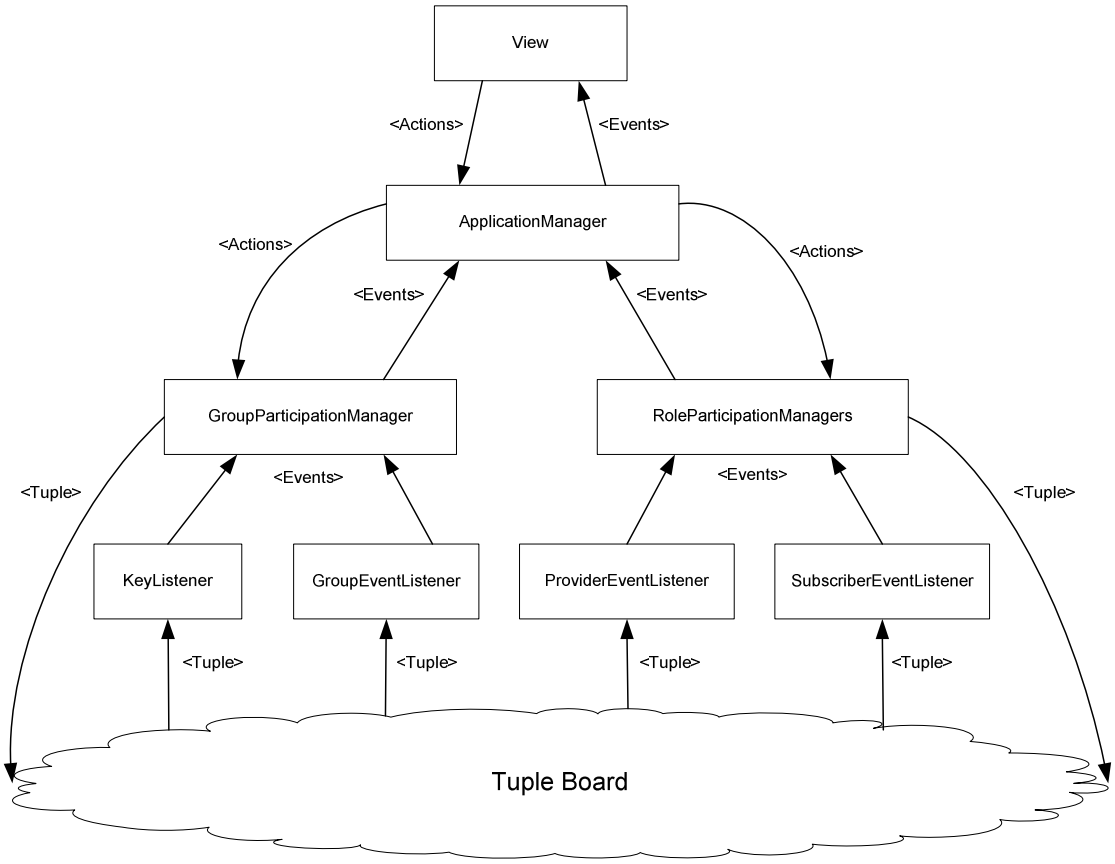


Fig. 11 – Application Architecture

The bottom tier consists of instances of *TupleListeners* which are added to the tuple board during application initialization. These listeners, mentioned in the protocols section above notify the participation manager tier of the posting and withdrawing of protocol tuples.

The participation manager tier responds to events from the listener tier and provides abstraction from the tuple board for the application manager tier when executing application protocols such as messaging. It also passes notifications of protocol activity up to the application tier through a custom event model.

The application tier passes actions from the view tier to the participation manager tier and also passes events received from the participation managers to the view. Each of these tiers is discussed in detail next.

Listener Tier

The listener tier is responsible for monitoring the tuple board and notifying the participation manager tier when tuples are posted and withdrawn. The implementation of this tier deviates slightly from the architectural diagram in that the listeners are actually implemented as anonymous inner classes contained within the participation manager tier. Seven different listeners are supported.

The first is a *PostListener* which listens for *GroupExistenceTuples*. The template tuple used is simply a blank *GroupExistenceTuple*. When matching tuples are posted this listener notifies its associated participation manager to initiate a key refresh.

The next two listeners are *PostWithdrawListeners* and perform the reading portion of the user announcement protocol described earlier. The subscriber listener uses a blank *SubscriberTuple* in order be notified when any subscribers come online or go offline. Similarly, the provider listener uses a blank *ProviderTuple* in order to be notified when any providers come online or go offline.

In order to implement the messaging protocol two additional *PostListeners* are defined. The group message listener listens for incoming group messages using a blank *GroupMessageTuple* template. The private message listener listens for incoming private messages using a *PrivateMessageTuple* template initialized with the user's *AlgorithmWalletCertificate* as the *to* field.

All of the listeners defined so far are contained within the *GroupParticipationManager*, a part of the participation manager tier responsible for all general group protocol activities. The remaining two listeners are unique for each role for which the application user is authorized. Together they implement the song transfer protocol.

The subscriber listener is only enabled if the current user is authorized as a subscriber. This listener listens for any incoming song transfers using a *SongTransferTuple* template initialized with the user's *AlgorithmWalletCertificate* as the *target* field. Similarly, the provider listener is only enabled if the current user is authorized as a provider. This listener listens for any incoming download requests using a *DownloadRequestTuple* template initialized with the user's *AlgorithmWalletCertificate* as the *providerCert* field. These listeners are contained with in role participation managers, the *SubscriberParticipationManager* and *ProviderParticipationManger* respectively.

Participation Manager Tier

The next application tier is the participation manager tier. In addition to containing the listener tier, this tier provides abstraction from the tuple board to the rest of the application. Notifications received from listeners are relayed through an event model to the application tier. In the opposite direction, actions requested in the application tier are delegated to this tier and translated into the necessary tuple protocol actions. Below are the classes that make up this tier.

Fig. 12 – Participation Manager Tier

The base *PartitionManager* class provides some functionality common to all types of participation managers in the tier. It provides members for the instance of *TupleBoard* being used, a queue of event listeners to notify of manager events and a collection of tuples that have been posted. The methods *postTuple()* and *withdrawTuples()* provide an easy way to track and remove tuples from the board for all managers. In addition it provides an *isAuthentic()* method which verifies signatures on tuples received from listeners. Since each manager is associated with a particular type of event listener from the model, generic *addListener()* and *removeListener()* methods are provided for modifying the listener queue for each manager.

Within the tier there are two different types of managers. The first type includes the *GroupParticipationManager,* which provides functionality common to all group users. The second type is the *RoleParticipationManagers*, of which there are two concrete implementations – one for subscribers and one for providers. These managers respond to role specific actions and events.

The *GroupParticipationManager* is the heavyweight of the tier. This manager controls the secure messaging protocol, the key exchange protocol and the user announcement protocol.

The ability to send messages is provided through the *sendGroupMessage()* and *sendPrivateMessage()* methods. The key exchange protocol is executed behind the scenes without any interaction, although the application is notified through the event model when key refreshes occur. Similarly, user announcements are posted automatically when the manager starts.

The remaining two managers, *ProviderParticipationManager* and *SubscriberParticipationManager* implement the song transfer protocol. Class *ProviderParticipationManager* manages the operations reserved for those users with the provider role, namely transferring songs to subscribers. Similarly, class *SubscriberParticipationManager* manages the operations reserved for those users with the subscriber role, namely downloading songs from providers.

As mentioned previously the manager classes of this tier relay events received from their enclosed listeners to the rest of the application through a custom event model. Below is the design of the event listener interfaces which make up this model.

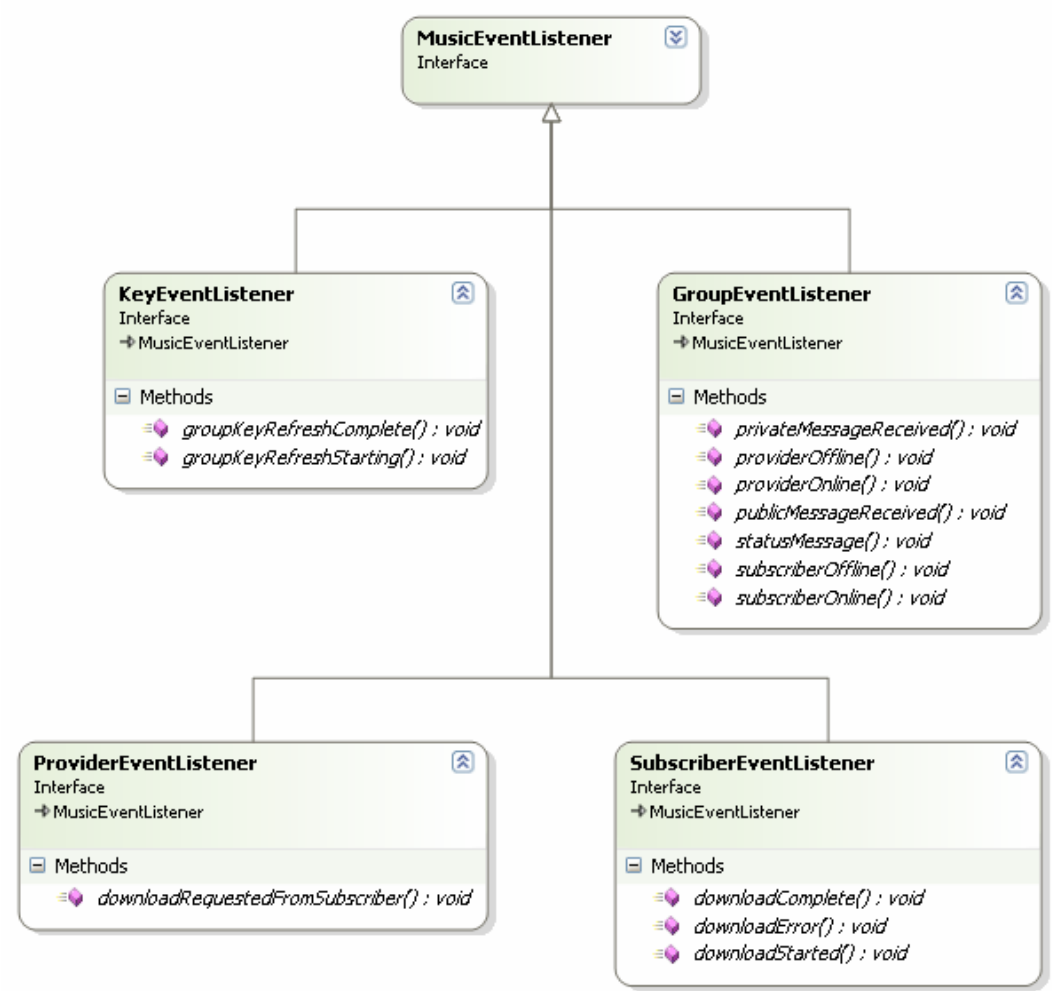


Fig. 13 – Participation Manager Tier Event Model

When protocol events occur in the manager tier they are relayed to the application through the event interfaces specified in the model. There is a one to one mapping between event listeners and participation managers. Class *GroupEventListener* specifies the events which are generated by the *GroupParticipationManager*. Classes implementing this event interface are notified when messages are received, when providers and subscribers come online or go offline, and when other status messages are generated by the manager. Class *ProviderEventListener* is a very simple listener, with only one event. This event is fired by the *ProviderParticipationManager* when a new incoming download request is received from a subscriber. Class *SubscriberEventListener* specifies events related to the download of songs by a subscriber. These events are fired by the *SubscriberParticipationManager* when a download is started, completed, or an error occurs during while executing the protocol.

Application Manager Tier

This tier acts as the controller between the view tier and the manager tier. Actions requested in the view are passed through this tier to the appropriate manager. In addition, events generated through the manager tier's event model are aggregated in this tier and relayed up to the view. This tier also handles most of the necessary security for the application. Below is a diagram of the classes which make up this tier.

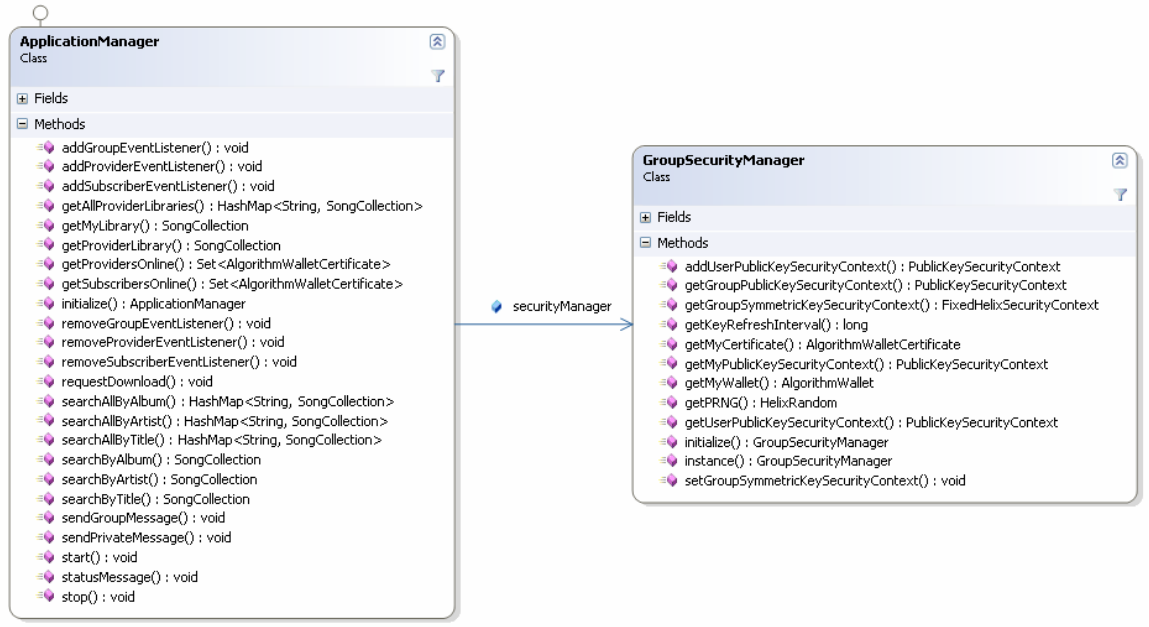


Fig. 14 – Application Manager Tier

Class *GroupSecurityManager* is implemented as a singleton and serves as a container for all security related processes in the application. This class holds a PRNG and the user's private algorithm wallet and public certificate as well as numerous security contexts. In addition to the group and user *PublicKeySecurityContexts* it also maps users to their respective *PublicKeySecurityContext* for easy lookup when communicating solely with a single user.

Class *ApplicationManager* servers as the main controller between the view and manager tiers. This class initializes all necessary participation managers when started and relays all events generated by them up to the view. The view may register listeners through the provided *add…Listener()* methods. In addition it provides methods for initiating all of the protocols supported by the system through the view. This class also represents a single point of storage for current group data.

Current collections of the subscribers and providers currently online are maintained as well as the available music libraries available by each provider and the application user. The manager also exposes music library search functionality, broken down by category and search breadth.

View Tier

The final tier in the application is the view. This tier allows interaction with the user, sends collected user actions to the application manager tier and displays events and notifications received from the lower tiers.

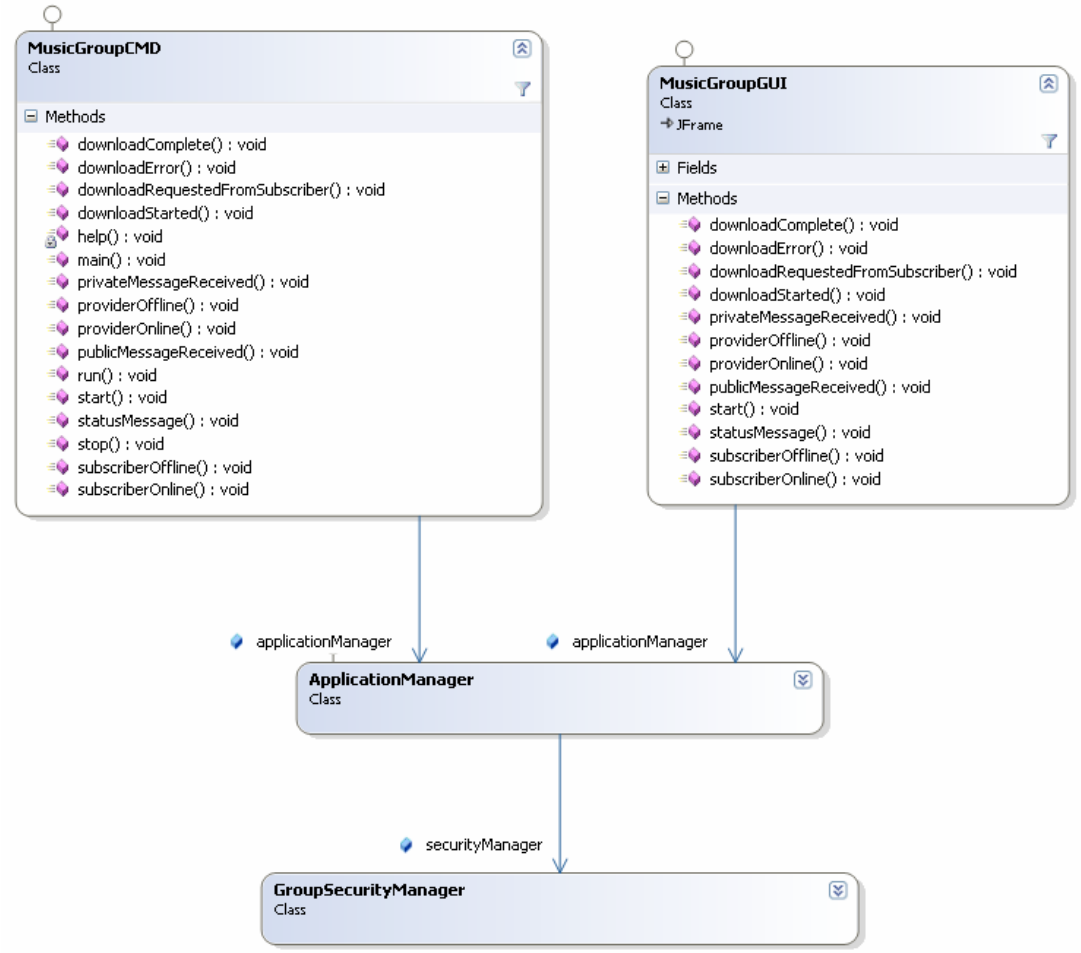


Fig. 15 – View Tier

Two different views compose this tier. Each view implements all event listener interfaces defined by the participation tier's event model. Both views also contain an instance of class *ApplicationManager* with which it registers itself as the necessary event listeners at startup.

The first view, a simple command line interface, is implemented in class *MusicGroupCMD*. In this view events received from the lower tiers are simply printed to standard out. Users may interact with the system via a predetermined set of commands. All basic application functionality is supported by this view, including sending and receiving group and private messages, downloading songs and searching.

The second view, implemented in class *MusicGroupGUI* uses Swing to provide a graphical interface. This view is much more involved than the simple command line interface and so requires some additional supporting classes, detailed below.

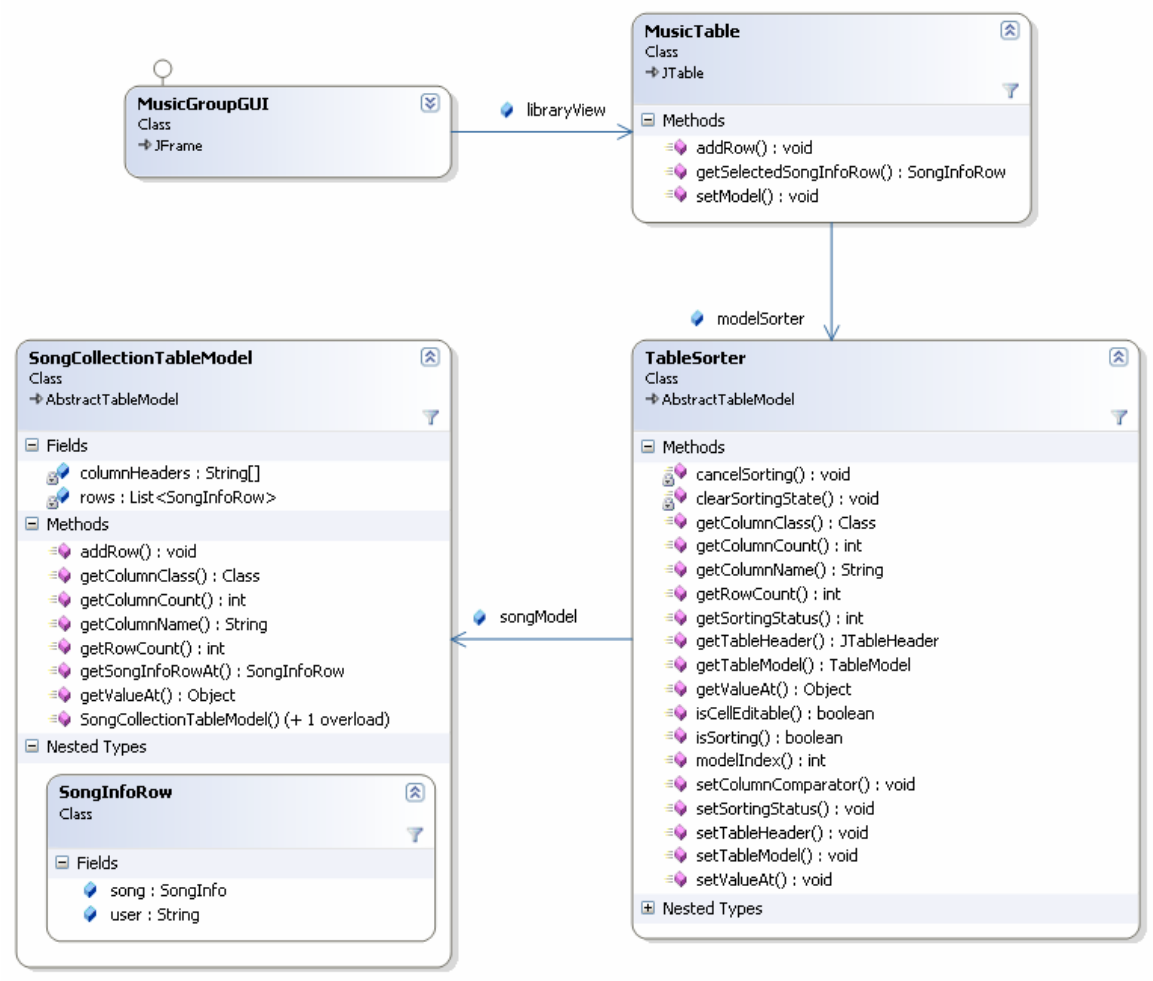


Fig. 16 – Graphical View Support

Class *MusicTable* provides a specialized extension of *JTable* suited for the display of song information and is used extensively through out the view. Class *TableSorter* provides the ability to sort these song information tables by column. It should be noted that this class was obtained from Sun's Swing Tutorials[6]. This functionality has since been absorbed into the Swing library with the recent release of Java6, but since this application is designed to be run with Java5 had to be manually included.

The final class, *SongCollectionTableModel* provides a custom data model for the *MusicTable* class to display. The data model allows the association with of a user with song information and the ability to display this data conveniently in the application.

For more information regarding usage of the application in each view mode see the User's Manual section found later in this document.

Performance Evaluation

Good security never comes for free. In order to establish the impact of encryption and authentication on performance in the library I created a small test program, *edu.rit.tb.music.test.PerformanceEval*. This program times how long each operation takes during the transfer of a *SongTransferTuple* from one device to another.

Usage:

```
%> java edu.rit.tb.music.test.PerformanceEval <certificate> <wallet> \
<file> <-provider | -subscriber> <none | auth | full>
```

The first two arguments specify the paths to the device's public *AlgorithmWalletCertificate* and private *AlgorithmWallet* respectively. The third argument specifies the path to the file to be transferred. The fourth argument specifies this process as the posting (-provider) or reading (-subscriber) process. Finally, the last argument specifies the security level of the transfer operation. A value of 'none' causes the tuple to be posted using the *DefaultSecurityContext* and to not be authenticated by the reading process. A value of 'auth' causes the tuple to still be posted using the *DefaultSecurityContext* but the signature is verified by the reading process. A value of 'full' causes the tuple to be posted using the *PublicKeySecurityContext* and the signature to also be verified by the reading process. Note that the posting process will post the *SongTransferTuple* and wait until killed externally while the reading process will perform one read, print the timing results to standard out and then exit.

All results were gathered running the posting process on *newyork* and the reading process on *georgia*, two machines on the CS network at RIT. Both machines contained one 650MHz SparcV9 processor and 512MB of ram and were running Solaris 10 at the time of testing. The machines were connected via a 100 Mbps Ethernet link. Three different files were transferred in this experiment:

File 1: 11 – the american analog set – aaron and maria.mp3 (3,045,376 bytes)
File 2: 04 – explosions in the sky – memorial.mp3 (12,775,424 bytes)
File 3: 03 – mono – yearning.mp3 (23,527,424 bytes)

Each song was transferred 100 times throughout the day at each security level (none, auth, full) in order to obtain these results.

Raw Transfer Time

This test was performed as a baseline with which to compare results of adding security features to the transfer. Here the song data is transferred in the clear and no authentication is performed by the reading process.

	File 1	File 2	File 3
Mean Transfer Time (msec)	2959.19	3840.89	4770.88

Fig. 17 – Mean Raw Transfer Times

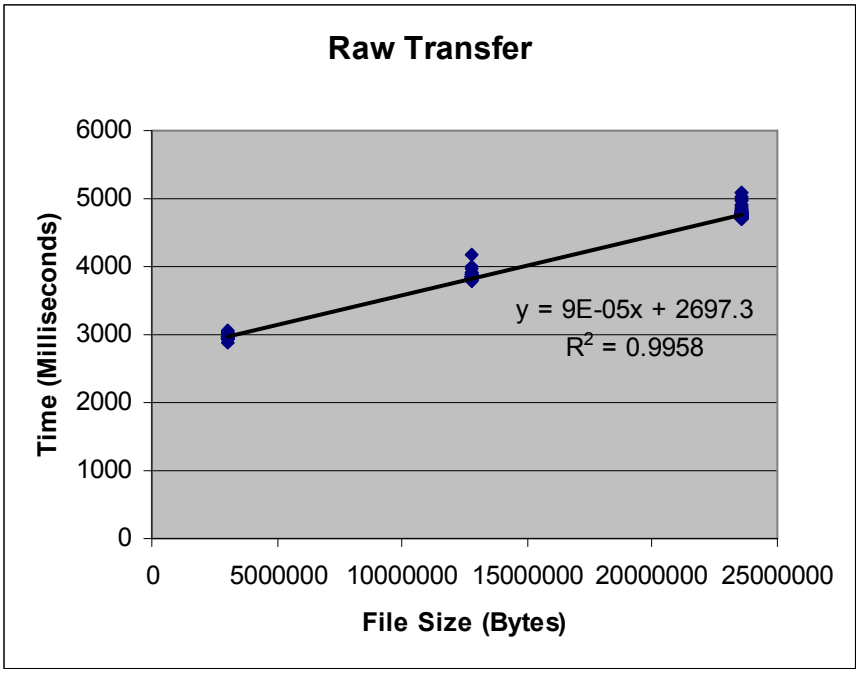


Fig. 18 – Raw Transfer Time

Signature Verification Time

This test was performed in order to determine how much time signature verification by the reading process contributes to the total transfer time. The signature algorithm used is 2048 bit RSA with Double SHA-256. Here the song data is transferred in the clear but the received *SongTranferTuple*'s signature is verified. Note that these timing measurements reflect only the time to verify the signature, not the time to transfer the data.

	File 1	File 2	File 3
Mean Signature Verification Time (msec)	868.57	3384.25	6144.92

Fig. 19 – Mean Signature Verification Times

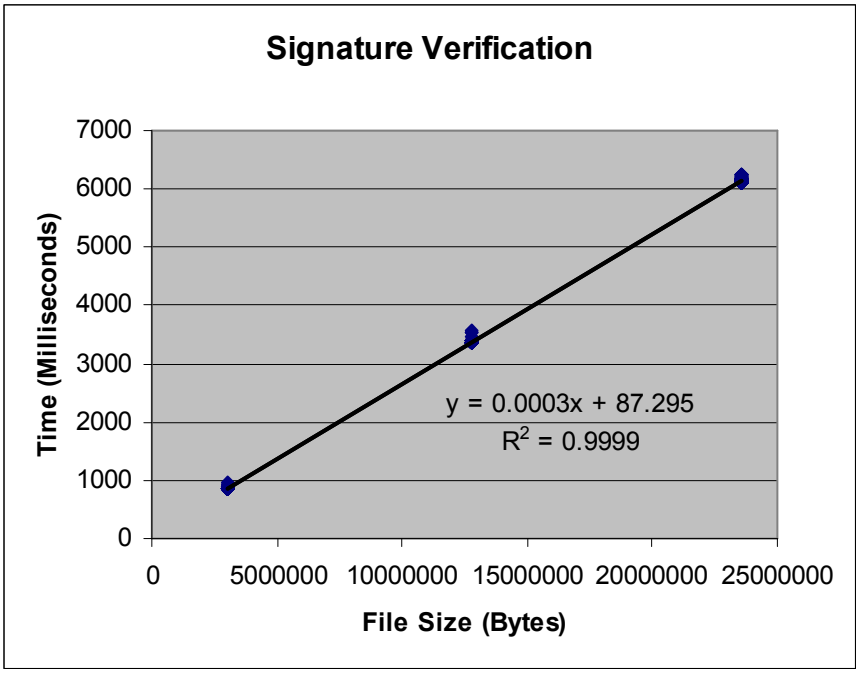


Fig. 20 – Signature Verification Time

Encrypted Transfer Time

This test was performed in order to determine how much time encryption and decryption during tuple transfer contributes to the total transfer time. The public key algorithm used is 2048 bit RSA. The symmetric key algorithm used is 256 bit Helix. No signature verification is performed by the reading process.

	File 1	File 2	File 3
Mean Encrypted Transfer Time (msec)	4812.44	7321.86	9920.45

Fig. 21 – Mean Encrypted Transfer Times

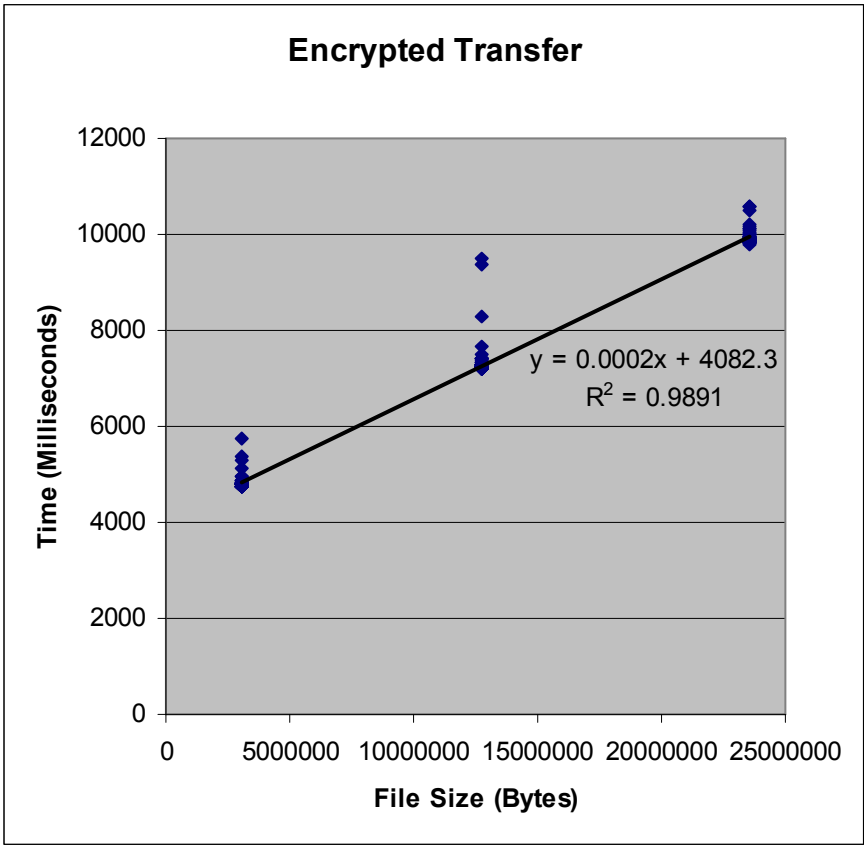


Fig. 22 – Encrypted Transfer Time

Authenticated & Encrypted Transfer Time

This test was performed in order to compare the total tuple transfer time using both encryption and authentication to the baseline raw transfer. The public key algorithm used is 2048 bit RSA. The symmetric key algorithm used is 256 bit Helix. The signature algorithm used is 2048 bit RSA with Double SHA-256.

	File 1	File 2	File 3
Mean A & E Transfer Time (msec)	5678.76	10756.11	16090.37

Fig. 23 – Mean Authenticated & Encrypted Transfer Times

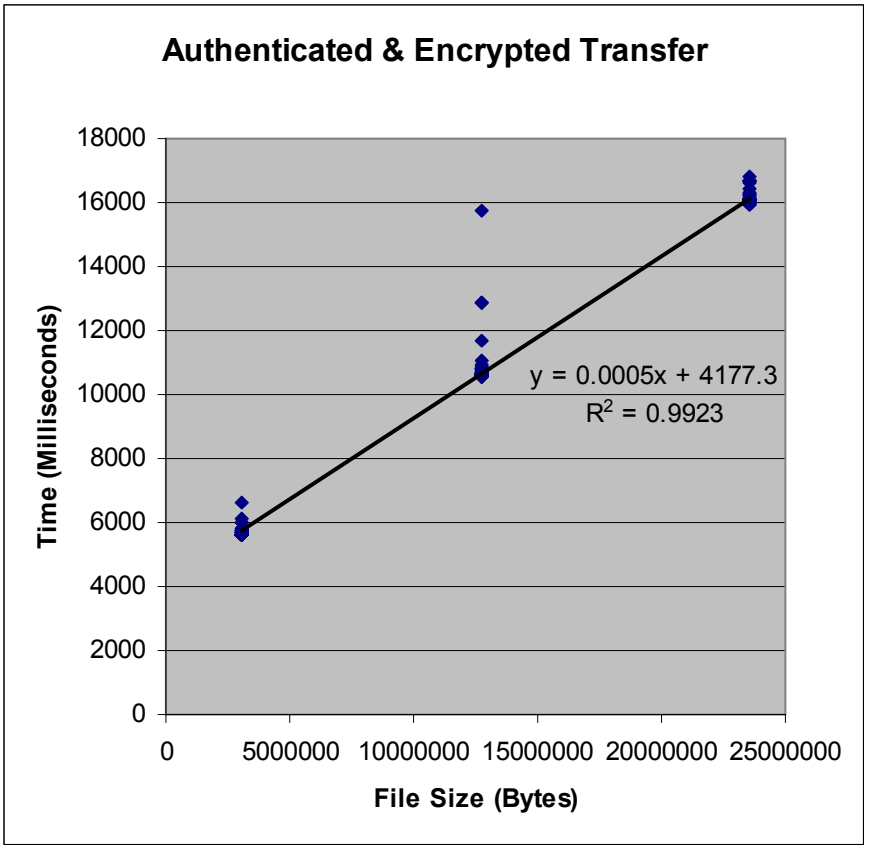


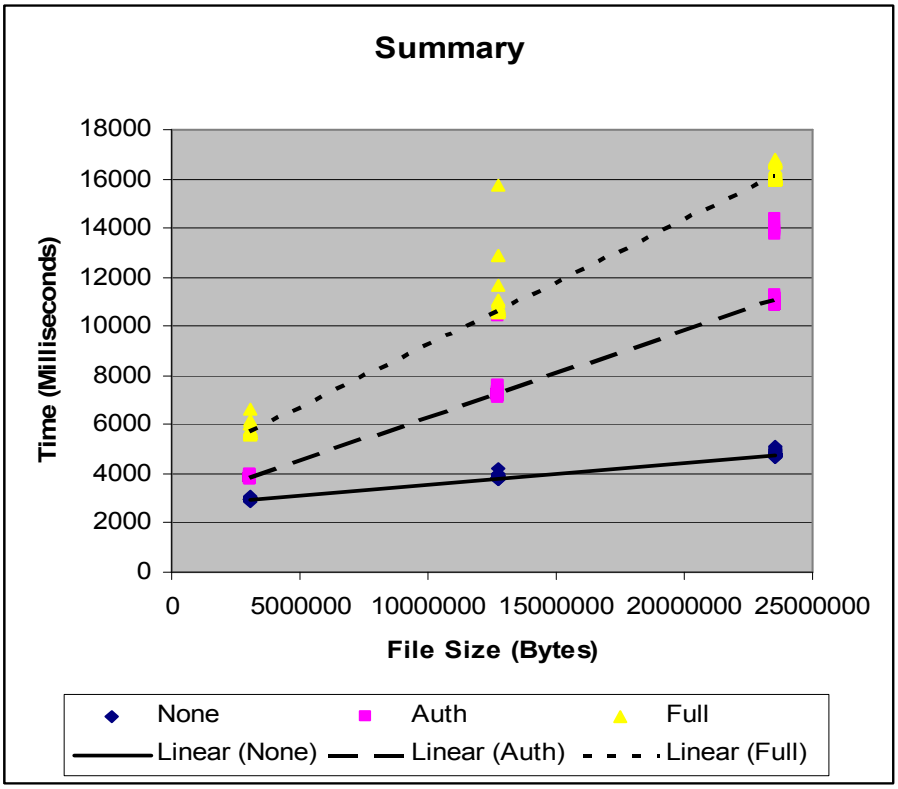
Fig. 24 – Authenticated & Encrypted Transfer Time

Summary

Once all testing data was gathered it could be condensed and analyzed for further performance comparison.

	None	Auth	Full
File 1	2959.19	3831.77	5678.76
File 2	3840.89	7261.05	10756.11
File 3	4770.88	11068.69	16090.37

Fig. 25 – Mean Total Transfer Times at Various Security Levels



None: $t = 0.00009B + 2697$ $R^2 = 0.9958$ Rate: 88.89 Mbps Overhead: 2697 msec
Auth: $t = 0.0004B + 2752$ $R^2 = 0.9791$ Rate: 20.00 Mbps Overhead: 2752 msec
Full: $t = 0.0005B + 4177$ $R^2 = 0.9923$ Rate: 16.00 Mbps Overhead: 4177 msec

Fig. 26 – Transfer Times & Rates at Various Security Levels

As expected, transfer times at all security levels increase linearly with the size of the file. In addition the rate of increase itself grows as authentication and encryption are added to the transfer. These results clearly illustrate the tradeoff between security and speed. It is therefore important to evaluate the environment in which the library is being used and select a security level appropriate for the desired level of performance.

User's Manual

The software described in this user's manual expects that certain jar files be on the class path in order to function correctly. Please refer to the Deliverables section found later in this document for more information regarding these required jars.

Required jar files:

- Latest TupleBoard API (tb.jar)
- Latest MusicGroup Application (musicgroup.jar)
- Java ID3 Tab Library (jid3lib-0.5.4.jar)
- NetBeans[11] Swing Absolute Positioning Library (AbsoluteLayout.jar, required only for graphical MusicGroup client)

The first entity which must be present in order to establish public key infrastructure is a certificate authority. In the TupleBoard API this is represented by class *edu.rit.tb.security.cert.CertificateAuthority*. This class comes equipped with a main method which enables creation of a CA and generation of certificates for distribution to devices.

Create a Certificate Authority

Before any certificates may be generated a static CA must first be created. The CA must be persistent across usage sessions because the signatures on all certificates must be verifiable by any device in the group regardless of when its certificate was generated. A certificate authority may be created by invoking the following command:

```
%> java edu.rit.tb.security.cert.CertificateAuthority –create \
<caName> <signatureAlgorithm> <encryptionAlgorithm> <bitStrength> \
<caFile>
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

`<caName>` - The name of the certificate authority. This should be unique as this name is also used internally as the group identifier during secure communication.

`<signatureAlgorithm>` - The name of the signature algorithm to use when signing certificates. The value of this parameter must be able to be parsed as an instance of the *edu.rit.tb.pki.PKIAlgorithmFactory.SignatureAlgorithms* enumeration. Currently supported values: { RSAWithDoubleSHA256}

`<encryptionAlgorithm>` – The name of the encryption algorithm to be used for group The value of this parameter must be able to be parsed as an instance of the *edu.rit.tb.pki.PKIAlgorithmFactory.SignatureAlgorithms* enumeration. Currently supported values: { RSA}

<bitStrength> – The bit strength of the algorithms. For the RSA encryption and signature algorithms this is the bit size of the public modulus, n. For RSA based algorithms this argument should be >= 2048.

<caFile> - The file to which the resulting *CertificateAuthority* object will be serialized for persistent storage.

The program will generate the public/private key pairs required for the signature and encryption algorithms and then serialize the resulting *CertificateAuthority* object to disk at the location specified. Generation of a 2048 bit RSA/RSAWithDoubleSHA256 certificate authority generally takes 8-10 minutes on a 650MHz SparcV9 CPU or 1 minute on a 1.8GHz Pentium M CPU.

Example:

```
%> java edu.rit.tb.security.cert.CertificateAuthority –create
groupOne RSAWithDoubleSHA256 RSA 2048 groupOne.ca

Creating CA signature public/private key pair...

Creating group encryption public private key pair...

Writing CA to /home/stu9/s15/krm4686/courses/msproj/data/groupOne.ca...

%>
```

Generate a Certificate

Once a certificate authority has been created device certificates may begin to be generated. In order to participate in secure group communication a device must have a valid certificate from the group CA. Device certificates are represented in the API by class *AlgorithmWalletCertificate*. A device certificate may be generated by invoking the following command:

```
%> java edu.rit.tb.security.cert.CertificateAuthority –generate \
<caFile> <startDate> <endDate> <id> <walletCertFile> \
<privateWalletFile>
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

<caFile> - The file in which the serialized contents of the certificate authority created in the previous step are stored.

<startDate> - The date this certificate becomes valid. Format: mm/dd/yyyy

<endDate> – The date this certificate expires. Format: mm/dd/yyyy

`<id>` – The ID of the user of device for which the certificate is being generated. This should be unique within the group.

`<walletCertFile>` - The file to which the resulting *AlgorithmWalletCertificate* object will be serialized for persistent storage.

`<privateWalletFile>` - The file to which the resulting *AlgorithmWallet* object will be serialized for persistent storage.

The program will deserialize the specified certificate authority and create a new *AlgorithmWallet*. This wallet, containing the private group public key encryption algorithm, user signature algorithm and user public key encryption algorithm, and an *AlgorithmWalletCertificate* created from the public instances of its algorithms are then serialized to disk at the locations specified. Generation of a 2048 bit RSA/RSAWithDoubleSHA256 certificate takes approximately 8-10 minutes on a 650MHz SparcV9 CPU or 1 minute on a 1.8GHz Pentium M CPU.

Example:

```
%> java edu.rit.tb.security.cert.CertificateAuthority –generate
groupOne.ca 06/07/2007 07/07/2007 user1 user1.cert user1.wallet

Creating user encryption & signature public and private key pairs...

Writing certificate to /home/stu9/s15/krm4686/courses/msproj/
data/user1.cert...

Writing wallet to /home/stu9/s15/krm4686/courses/msproj/
data/user1.wallet...

%>
```

Generate a Music Collection

Once a certificate and wallet have been generated as in the previous step the user is now capable of interacting with an arbitrary group. In order to participate in a music group however some additional certificates are required – that of a provider and subscriber. The first step in generating these is to create a *MusicCollection* object. This is a list of all artists, albums and songs a provider or subscriber is permitted to upload or download respectively. To generate a music collection:

```
%> java edu.rit.tb.music.meta.MusicCollection <albumsFile> \
<songsFile> <artistsFile> <outFile>
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

`<albumsFile>` - The text file containing authorized albums. Each line should be formatted so it may be parsed into an *AlbumInfo* object through the *AlbumInfo.fromAlbumInfoString()* method. Format: year;;;artistName;;;albumName

<songsFile> - The text file containing authorized individual songs. Each line should be formatted so it may be parsed into a *SongInfo* object through the *SongInfo.fromSongInfoString()* method. Format: year;;;artistName;;;albumName:::trackNumber:::title

<artistsFile> - The text file containing authorized artists. Each line should contain a single string representing the artist's name. Format: artistName

`<outFile>` - The file to which the resulting *MusicCollection* object will be serialized for persistent storage.

The program will parse the three input files into their appropriate collections (*AlbumCollection*, *SongCollection* and *ArtistCollection* respectively). A new *MusicCollection* object created from these collections is then serialized to disk at the specified location.

Example:

```
%> java edu.rit.tb.music.meta.MusicCollection albums.txt songs.txt
artists.txt user1.mc

Parsing album collection...

Parsing song collection...

Parsing artist collection...

Writing music collection to /home/stu9/s15/krm4686/courses/msproj/
data/user1.mc...

%>
```

Generate a Music Authorizations Certificate

Now that a music collection has been generated certificates may be created for participation in the music group. Class *edu.rit.tb.music.cert.MusicCertificateAuthority* contains a main method capable of generating provider and subscriber certificates using the music collection created in the previous step and the certificate authority created a few steps back. *MusicAuthorizationCertificate*s for both roles are created in the following manner:

```
%> java edu.rit.tb.music.cert.MusicCertificateAuthority –<role> \
<caFile> <startDate> <endDate> <id> <musicCollectionFile> \
<ouputCertFile>
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

`<role>` - The role granted by this certificate. Must be one of 'provider' or 'subscriber'.

`<caFile>` - The file in which the serialized contents of the *CertificateAuthority* are stored.

`<startDate>` - The date this certificate becomes valid. Format: mm/dd/yyyy

`<endDate>` – The date this certificate expires. Format: mm/dd/yyyy

`<id>` - The ID of the user of device for which the certificate is being generated. This must match the ID contained in the user's *AlgorithmWalletCertificate*.

`<musicCollectionFile>` - The file in which the serialized contents of the *MusicCollection* object created in the last step are stored.

`<outputCertFile>` - The file to which the resulting *MusicAuthorizationCertificate* object will be serialized for persistent storage.

The program will deserialize the specified CA and music collection objects from disk and then create and sign a new *MusicAuthorizationCertificate* for the specified role. The certificate is then serialized to disk at the specified location.

Example:

```
%> java edu.rit.tb.music.cert.MusicCertificateAuthority -subscriber
groupOne.ca 06/07/2007 09/07/2007 user1 user1.mc user1.sub

Loading certificate authority...

Loading music collection...

Creating certificate...

Writing certificate to /home/stu9/s15/krm4686/courses/msproj/
data/user1.sub...

%>
```

Join a Music Group (Command Line Client)

Once all the previous steps have been completed it will be possible to run the MusicGroup application. There are two versions of the application – a text based command line client and a Swing based graphical client. This section describes how to use the command line version.

```
%> java edu.rit.tb.music.MusicGroupCMD <userID> <groupID> <certDir> \
[-provider <musicLibDir>] [-subscriber <musicLibDir>]
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

`<userID>` - The unique identifier of the user participating in the group.

`<groupID>` - The identifier of the group to be joined. This corresponds to the name of the CA which generated the specified user's certificates.

`<certDir>` - The directory containing all relevant certificate files necessary for group participation.

`[-provider <musicLibDir>]` – This argument set specifies that the application enable the provider role. All valid *MusicAuthorizationCertificate* files located in the certificate directory will be loaded. All authorized song files located in the specified music library directory or any of its subdirectories will be added to the provider's list of available songs.

`[-subscriber <musicLibDir>]` – This argument set specifies that the application enable the subscriber role. All valid *MusicAuthorizationCertificate* files located in the certificate directory will be loaded. All song files located in the specified music library directory or any of its subdirectories will be added to the subscriber's current media list.

Note: At least one of these role argument sets must be specified. It is also valid to specify both.

On start up the application will search for and deserialize the user's certificate and wallet files, enable the specified role(s), load each role's song collection metadata, and join the group. It will then prompt for input and respond to user commands as they are entered. For additional details on the protocol for joining a group and other group interaction including sending messages and transferring files please refer to the MusicGroup Application Design & Implementation section, found earlier in this document.

Example:

```
%> java edu.rit.tb.music.MusicGroupCMD user1 groupOne data/
-provider music/ -subscriber music/
```

```
[Initializing application manager...]

[Reading existing group data...]

[No existing group data found.  Creating new group...]

[Subscriber online: user1]

[Provider online: user1]

> _
```

The application is now fully initialized and ready to process user commands.

System Messages

As the command line client is being used various system messages will be displayed to the console. All system messages are contained within square brackets like this:

[system message]

Examples of system messages include provider and subscriber online and offline notifications, download requests, group key exchange information and incoming private and group messages.

Available Commands:

The list of commands the application responds to may be accessed by typing 'help'

```
> help

Available commands:
==========================

download <provider> \                          Download the specified song
       "<artist - title>" "<path>"

gm <message>                                   Send a message to all group users

help                                           Print this menu

list <all | my | providerID>                   View available song library

online <providers | subscribers>               View users currently online

pm <id> <message>                              Send a message to user 'id'

quit                                           Quit the application

search <all | providerID> \                    Search function
       <artist | album | title> \
       <string>
```

Who's Online?

A good place to start when using the application is to see what other subscribers and providers are online. While system messages are generated each time a new user enters or leaves the group it is sometimes convenient to get the list of all currently participating users. To do so use the 'online' command:

```
> online subscribers

Subscribers online:
====================
user1
user2
user3
user4

> _
```

The same syntax is used to view the list of providers currently online by replacing 'subscribers' with 'providers'.

```
Providers online:
==================
user1
user2
user3
user4

> _
```

What Songs are Available?

Since this is a music sharing client it is useful to be able to browse the available catalogs for each provider currently online. This functionality is implemented through the 'list' command. There are three different versions of this command.

To view the songs currently available from all providers in the group:

```
> list all

user1's library:
=================
the american analog set – aaron and maria
mono – the flames beyond the cold mountain

user2's library:
=================
explosions in the sky – first breath after coma
mono – yearning

> _
```

To view the songs currently available from a specific provider:

```
> list user2

user2's library:
================
explosions in the sky – first breath after coma
mono – yearning

> _
```

To view the songs in my library:

```
> list my

user1's library:
================
the american analog set – aaron and maria
mono – the flames beyond the cold mountain

> _
```

Music Search

Manually browsing through the output of the 'list' command can become tedious, especially if providers have large libraries or there are many providers currently online. The 'search' command allows querying for specific albums, artists or songs from specific providers or the entire group based on the command syntax chosen. Result matching is performed on a substring basis only. Songs are determined to match the query if the song field (album, artist or title) contains the query or vice versa.

To search all providers for a song by title:

```
> search all title flames beyond the cold mountain

user1's library:
================
mono – the flames beyond the cold mountain

> _
```

To search a specific provider for a song by title:

```
> search user2 title flames beyond the cold mountain

No results found

> _
```

The previous two queries searched for matches by title. Artists or albums may also be queried by replacing the 'title' keyword with 'artist' or 'album' respectively.

```
> search all artist explosions

user2's library:
================
explosions in the sky - first breath after coma

> _
```

Messaging

The application supports two types of messaging protocols. Group messages are able to be received by each user currently participating in the group. In contrast, private messages are only able to be read by their target recipient. Incoming messages are relayed through the system message queue as described earlier in this section.

To send a group message use the 'gm' command:

```
> gm this is a public message

> _
```

That command would cause the following system message to appear on each group user's console:

```
[Public message from user1: this is a public message]
```

To send a private message use the 'pm' command:

```
> pm user2 this is a private message to user2

> _
```

That command would cause the following system message to appear on user2's console:

```
[Private message from user1: this is a private message to user2]
```

Downloading Songs

No file sharing application would be a file sharing application without the ability to download songs. Once a song has been found through either the 'list' or 'search' commands it may be downloaded by using the 'download' command.

```
> download user2 "mono – yearning" "/tmp/downloads/mono-yearning.mp3"

> _
```

Note the use of quotes to enclose the song to be downloaded and the path to which the resulting file will be saved. These are required for the command to succeed. The above command will initiate a download request with the specified provider for the specified song. Provider user2 will receive the following system message:

[Subscriber user1 requesting download of mono – yearning]

The following system messages will be displayed on subscriber user1's console:

[Download started: mono – yearning]

[Download completed: /tmp/downloads/mono-yearning.mp3]

The above exchange assumes that subscriber user1 is authorized to download the song "mono – yearning" and that provider user2 is authorized to provide it. If either of these are not the case error messages will be received instead.

Quitting the Application

When finished interacting with the music group simply use the 'quit' command.

```
> quit

%>
```

All tuple board interactions will be immediately ceased and the application will shutdown, returning to the shell from which it was started.

Join a Music Group (Graphical Client)

This section describes how to use the Swing based graphical Music Group client. The syntax of this program is identical to that of the command line version.

```
%> java edu.rit.tb.music.MusicGroupGUI <userID> <groupID> <certDir> \
[-provider <musicLibDir>] [-subscriber <musicLibDir>]
```

Note: '\' denotes a line continuation; there should be no newlines in the argument list.

Argument descriptions:

All arguments are identical to those for the command line client. Please see the previous section for descriptions on the arguments required for the application.

Example:

```
%> java edu.rit.tb.music.MusicGroupGUI user1 groupOne data/
-provider music/ -subscriber music/
```

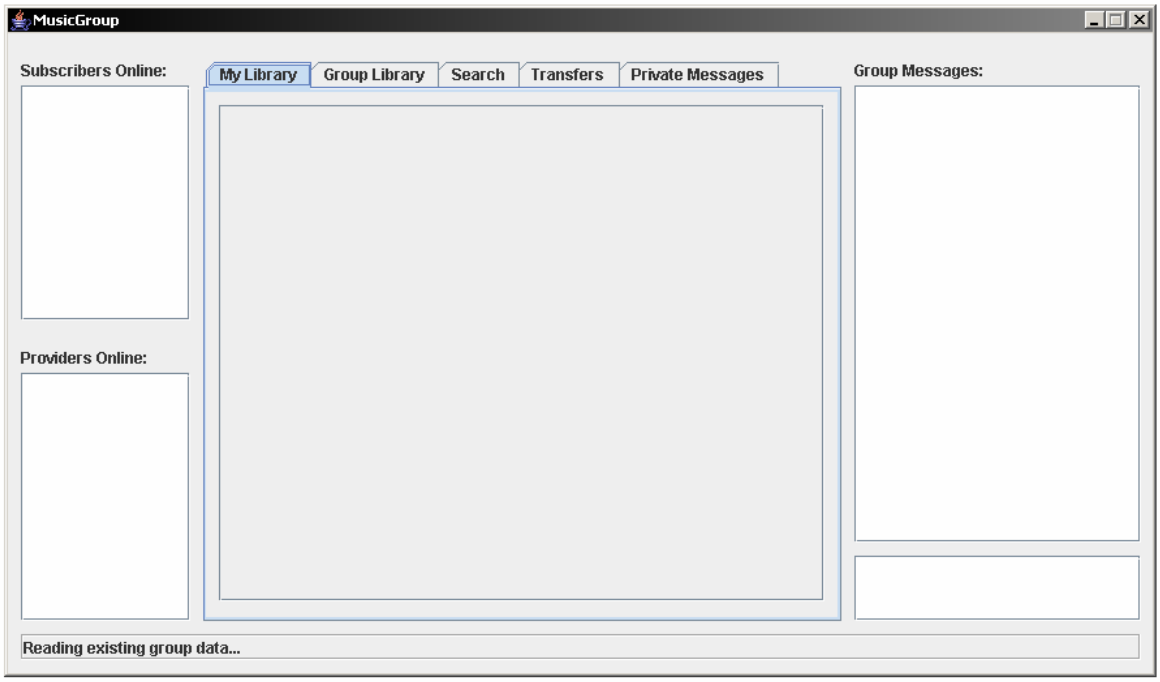


Fig. 27 – MusicGroup Graphical Client

The graphical client will perform all the same steps as the command line client on startup - deserialize the user's certificate and wallet files, enable the specified role(s), load each role's song collection metadata, and join the group.

System Messages

As the graphical client is being used various system messages will be displayed in the status bar area at the bottom of the main window.

Examples of system messages include provider and subscriber online and offline notifications, download requests, group key exchange information and incoming private and group messages.

Who's Online?

A good place to start when using the application is to see what other subscribers and providers are online. While system messages are generated each time a new user enters or leaves the group the current group members are also shown in the left zone. Current subscribers appear in the top list, labeled 'Subscribers Online:'. Similarly, current providers appear in the bottom list, labeled 'Providers Online:'.

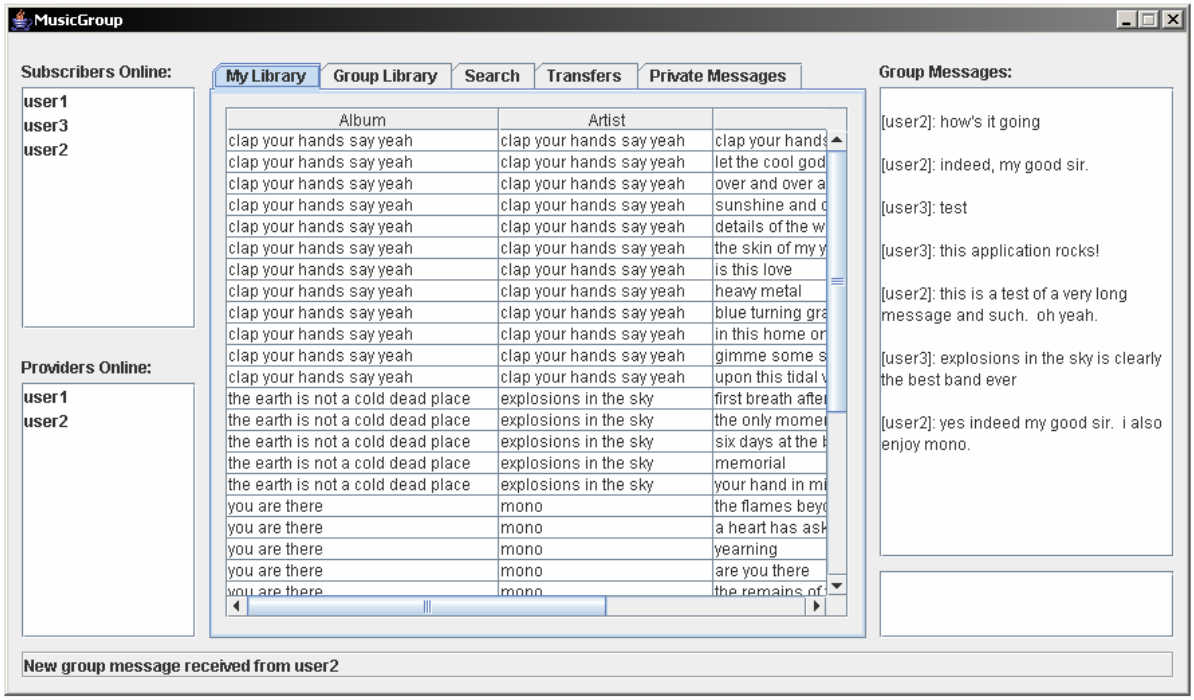


Fig. 28 – Current users online appear in the left zone

What Songs Do I Have?

Since this is a music sharing client it is useful to be able to see what songs are already in the user's library.

The current user's library may be viewed from the 'My Library' tab in the center zone, pictured below.

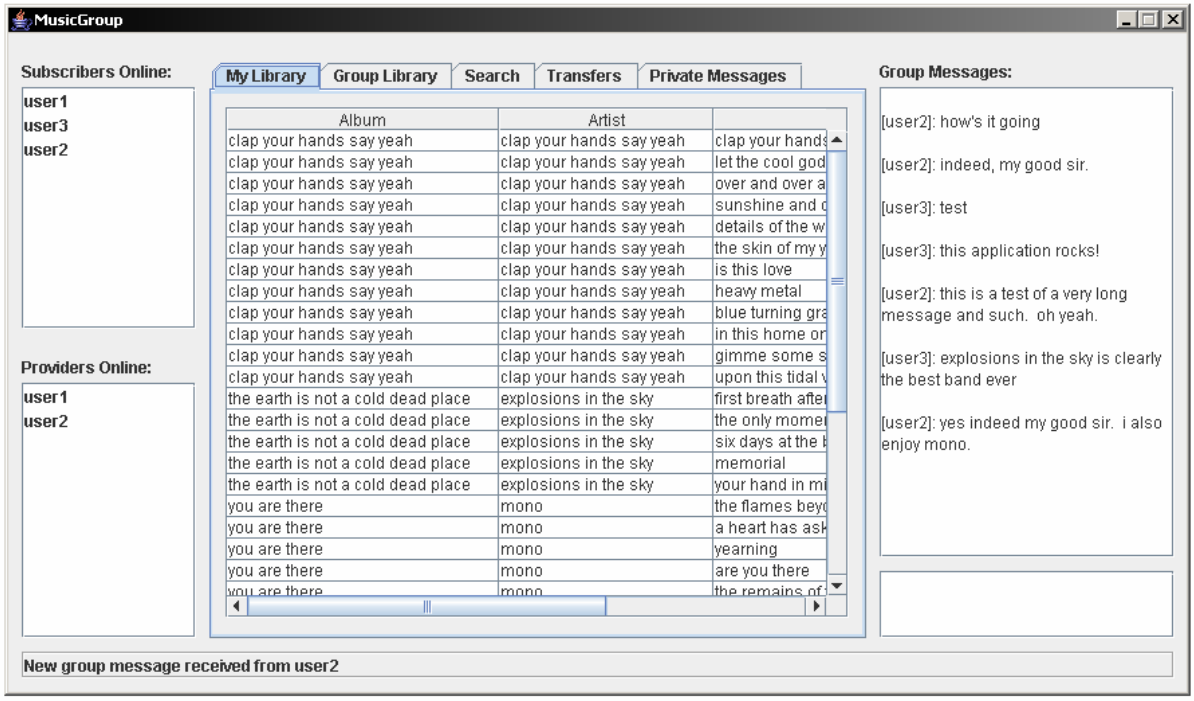


Fig. 29 – My Library

This tab contains a list of all songs loaded from any `<mediaLibrary>` arguments found on the command line at application start. (See the beginning of this section for more information on arguments). This view is analogous to the 'list my' command in the command line version of the client.

Song tables like the one pictured above occur throughout the various tabs in the application. All tables may be sorted by clicking on the column header which corresponds to the data field to be sorted on. Clicking once will sort the data in ascending order. Clicking again will sort the data in descending order. Clicking once more will reset the data to its original ordering.

What Songs Are Available?

Even more useful is the ability to browse the available catalog for each provider currently online. The complete list of all available songs and their providers may be viewed by clicking the next tab to the right of 'My Library', labeled 'Group Library'. This view is analogous to the 'list all' command in the command line version of the client.

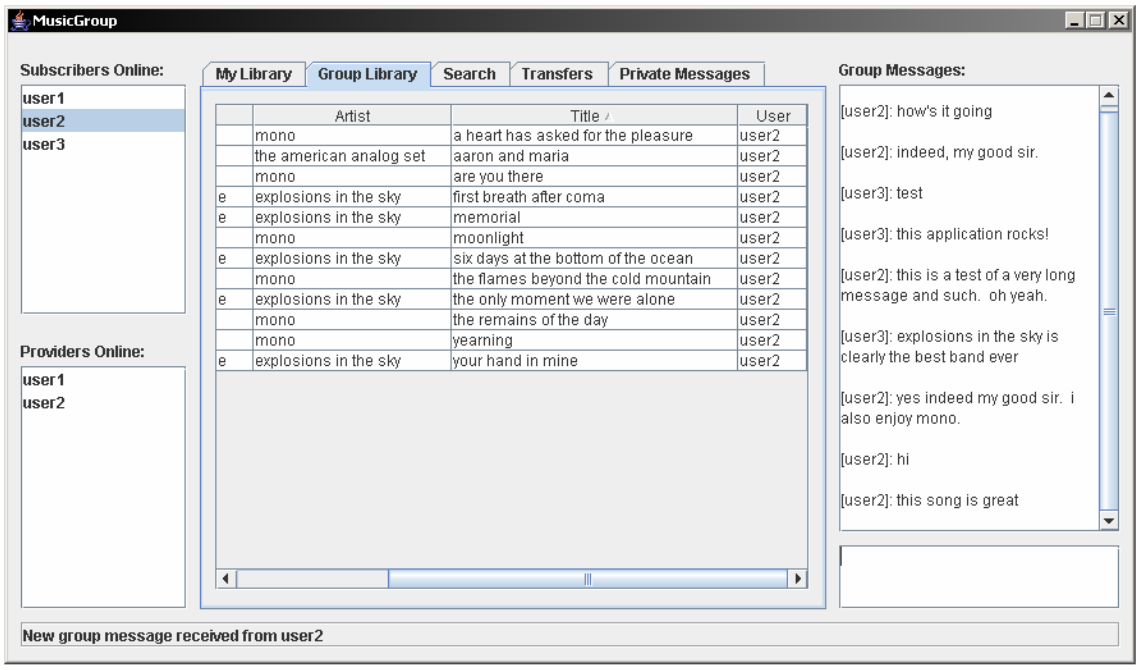


Fig. 30 – Group Library

Here we can clearly see all the songs available for user2, the only other provider currently online. Note that if the current user is also a provider the corresponding library is excluded from this view since all entries will already be shown in the 'My Library' view and secondly it would not make sense to download a song from oneself. The above view has also been sorted by title in ascending order to illustrate the previously mentioned sorting abilities.

Music Search

Manually browsing through the 'Group Library' view can become tedious, especially if providers have large libraries or there are many providers currently online. The search interface, located in the next tab to the right, labeled 'Search', allows querying for specific albums, artists or songs from specific providers or the entire group based on the options chosen. Result matching is performed on a substring basis only. Songs are determined to match the query if the song field (album, artist or title) contains the query or vice versa. This view is analogous to the 'search' command in the command line client.

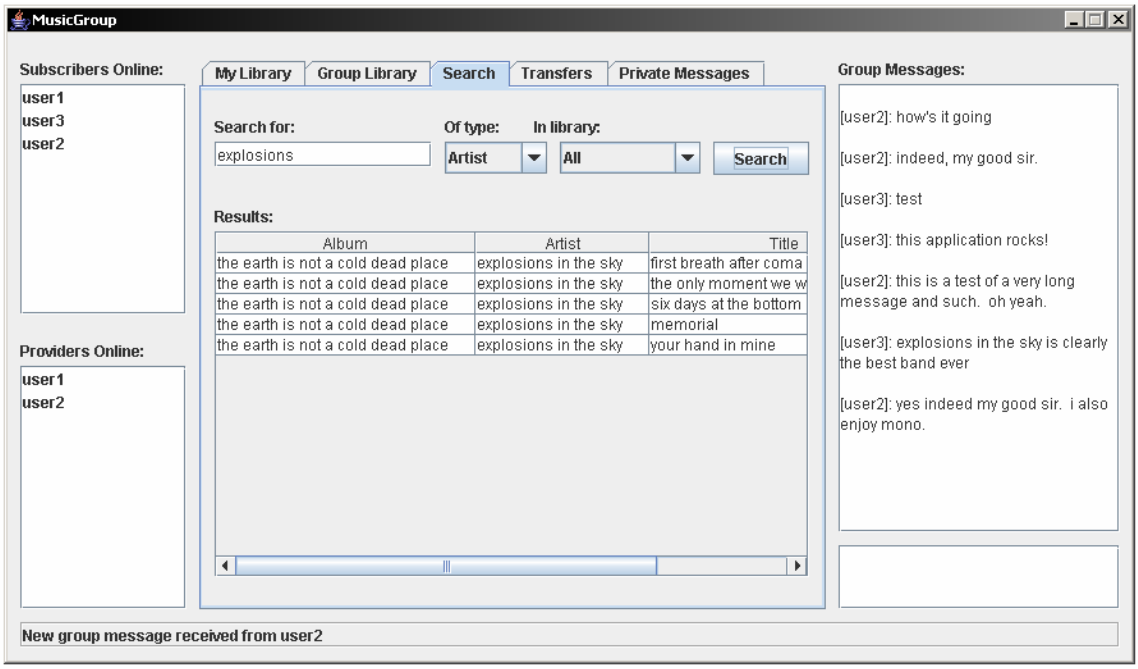


Fig. 31 – Search

Here we can see the results of searching for the term 'explosions' in the artist field of all songs in all provider's libraries. Queries may also be performed against the title or album fields by selecting the appropriate value in the first drop down box. In addition it is possible to search only a single provider rather than all of them. To do so, simply select the desired provider to be searched from the second dropdown box. When the 'Search' button is clicked the search is performed, any old results are cleared from the results view and the new results are loaded. If no results match the specified query the results view will simply remain empty.

Downloading Songs

No file sharing application would be a file sharing application without the ability to download songs. Song downloads may be initiated from both the Group Library and Search views. In the Group Library simply double click the table row containing the desired song and provider. After a search simply double click the results table row containing the desired song and provider. This action will launch a download prompt similar to that pictured below. This behavior is analogous to invoking the 'download' command in the command line client.

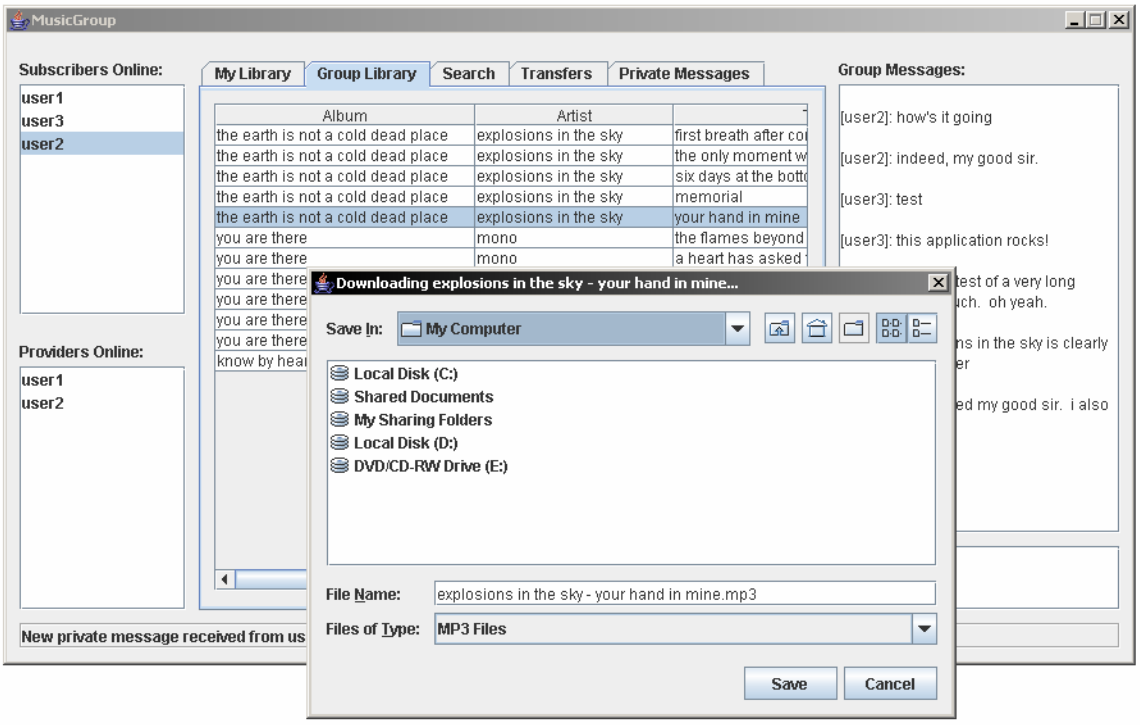


Fig. 32 – Download Prompt

Select a suitable place to save the file click the 'Save' button to initiate the download request. System messages will be displayed in the status bar area both when the transfer begins and when it has completed:

Download started: explosions in the sky – your hand in mine

Download completed: <path to file>/explosions in the sky – your hand in mine.mp3

The above exchange assumes that subscriber user1 is authorized to download the song "explosions in the sky – your hand in mine" and that provider user2 is authorized to provide it. If either of these are not the case error messages will be received instead.

Song Transfer Log

A feature unique to the graphical client is the song transfer log. This view functions as a song transfer history and lists all files which have been uploaded or downloaded during the current session. The log is accessible through the fourth tab in the center zone, labeled 'Transfers'.

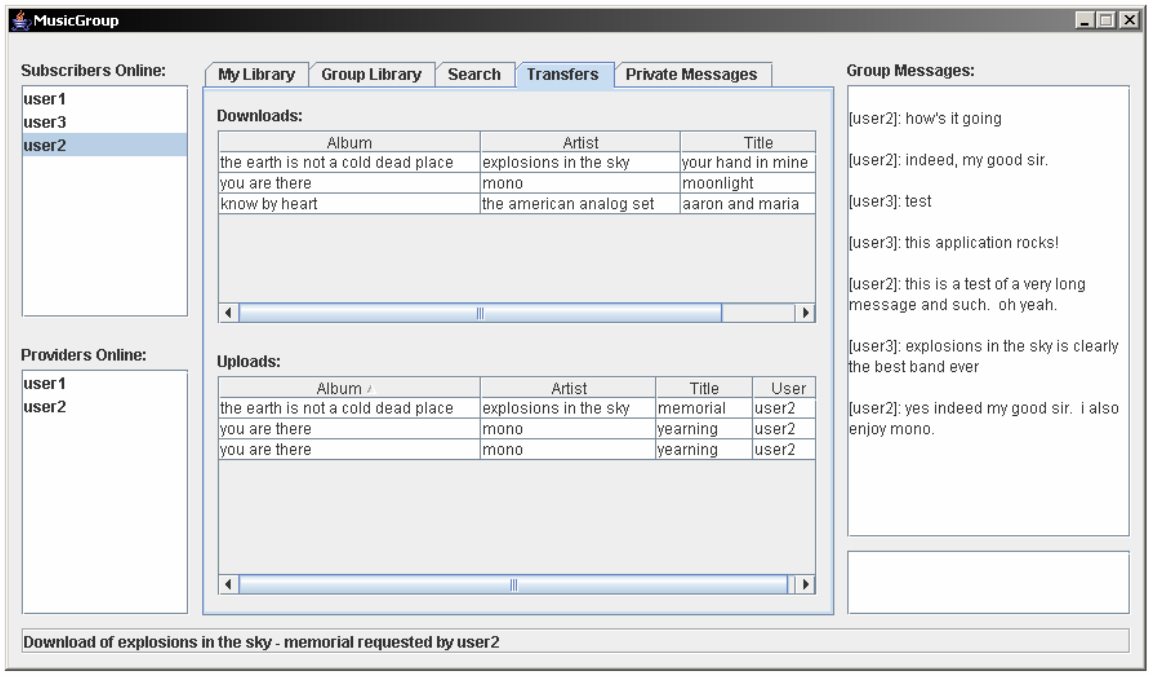


Fig. 33 – Song Transfer Log

Transactions are broken down into downloads and uploads. Each list contains an entry for each song transferred and the user at the other end of the exchange. Note that if the current user does not have the subscriber role enabled the download log will be empty. Similarly if the current user does not have the provider role enabled the upload log will be empty.

Messaging

Like the command line client, the graphical client supports two types of messaging protocols. Group messages are able to be received by each user currently participating in the group. In contrast, private messages are only able to be read by their target recipient. Incoming messages are relayed through the system message queue and displayed in the status bar area as described earlier in this section.

The graphical client also additional interfaces for sending and presenting received messages, both public and private. Private messages may be sent and received from the last tab in the center zone, labeled 'Private Messages'.

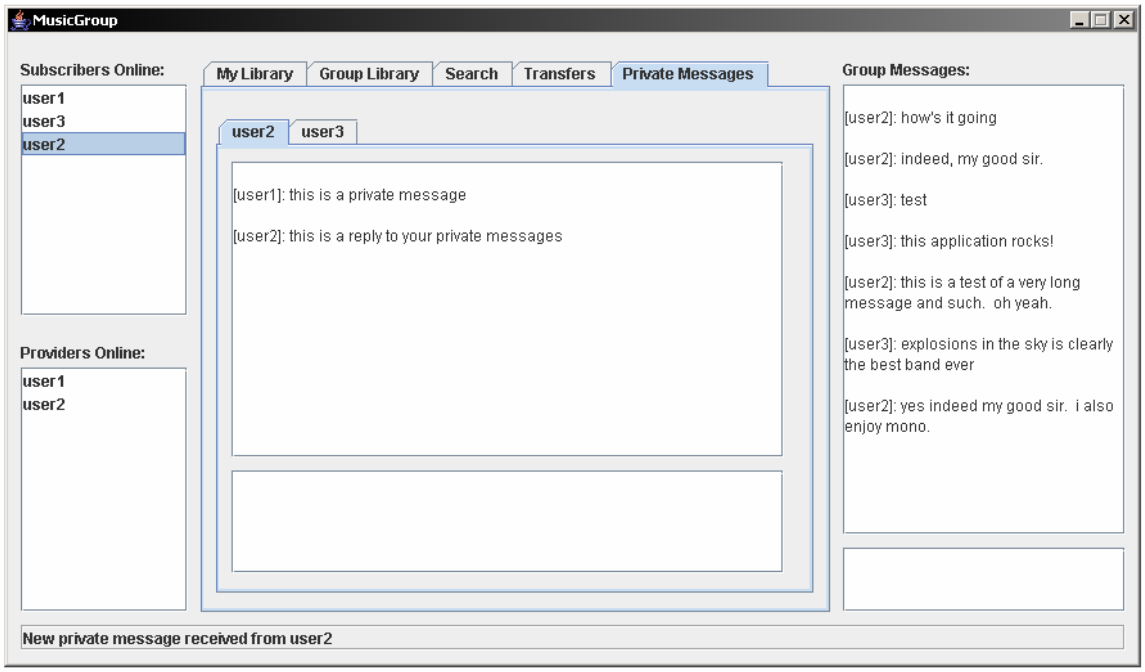


Fig. 34 – Private & Group Messages

Private conversation windows like the one pictured above are automatically created in the 'Private Messages' tab when a new private message is received from another user in the group. To initiate a new private conversation with another group user simply locate the user's ID in one of the online user lists in the left zone and double click. The private message interface will automatically gain focus and a new conversation window will be created for the target user. Sent and received messages are logged in the top text area, while messages to be sent are entered in the bottom text area. The message is sent once the 'Enter' key is typed. This behavior is analogous to the 'pm' command in the command line client.

Group messages, or public messages as they are also known are sent and received in the 'Group Messages' pane in the right and final zone. The behavior of the group message interface is identical to that of the private message conversation windows. Sent and received messages are logged in the top text area, while messages to be sent are entered in the bottom text area. The message is sent once the 'Enter' key is typed. This behavior is analogous to the 'gm' command in the command line client.

Quitting the Application

When finished interacting with the music group simply close the application window. All tuple board interactions will be immediately ceased and the application will shutdown, returning to the shell from which it was started.

Developer's Guide

With the many additions made to the TupleBoard API during the project it is useful to provide some developer's suggestions and small code samples demonstrating how to get the most out of the new security functionality.

Working with Random Numbers

As mentioned in the section detailing the design and implementation of the newly added Helix cipher based pseudorandom number generator *HelixRandom*, a PRNG's data is only as good as its seed. When working with this class it is important to initialize it with a proper, truly random seed.

Fortunately, Java5 makes it fairly easy to do so. Class *java.security.SecureRandom* provides the ability to generate seed bytes to be used with other PRNGs. The underlying manner in which these seed bytes are created differs from system to system. In Linux and UNIX based environments there is an entropy source provided at the OS level, through the */dev/random* or */dev/urandom* devices. The Windows platform has no analogous file, but enables randomness to be obtained through the native CryptoAPI[10]. Java uses a system property, *securerandom.source* to control the entropy source when generating seeds. As long as the value of this property is set to "file:/dev/urandom" Java will automatically use the correct entropy source for the environment. This is the default value.

Using *HelixRandom* is easy. The following code sample illustrates initializing a new instance of the PRNG with seed bytes obtained as describe above and then generating 100 random bytes.

```
//Initialize seed generator
SecureRandom seedGen = new SecureRandom();

//Generate seed bytes
byte[] seedBytes = seedGen.generateSeed(48);

//Create new HelixRandom
HelixRandom helixRand = new HelixRandom(seedBytes);

//Now random bytes may be obtained
byte[] randomBytes = new byte[100];
helixRand.nextBytes(randomBytes);
```

Fig. 35 – Generating Randomness with HelixRandom

Working with Public Key Algorithms

Currently the TupleBoard library supports one public key encryption algorithm – RSA. Instances of the algorithm may be obtained through the provided algorithm factory class, *PKIAlgorithmFactory*. The factory automatically generates the necessary keys according to the bit strength specified. While any number of bits may be used, a value of 2048 is recommended as a minimum to achieve a reasonable level of security.

```
//Specify the bit strength of the algorithm to be created
int bitStrength = 2048;

//Obtain an instance of RSA
PublicKeyAlgorithm rsa = PKIAlgorithmFactory
                         .createEncryptionAlgorithm
                               (EncryptionAlgorithms.RSA,
                                bitStrength,
                                helixRand);
```

Fig. 36 – Obtaining an instance of RSA

Once an instance of RSA has been obtained it may be used for encryption. RSA operates on instances of *java.math.BigInteger* so a plaintext message must be represented as a number *p* such that *p* <= *n*, the public RSA modulus. This conversion is known as a padding scheme. Remember that RSA should not be used for bulk encryption operations, but instead for exchanging the key for the symmetric algorithm that will do the bulk encryption.

```
//Create the plaintext message
BigInteger plaintext = new BigInteger("1234567890");

//Encrypt the message
BigInteger ciphertext = rsa.encrypt(plaintext);

//Decrypt the message
BigInteger decrypted = rsa.decrypt(ciphertext);

//Verify the operation was a success
if(plaintext.equals(decrypted)) {
      System.out.println("Success");
}
```

Fig. 37 – Encryption and Decryption using RSA

Working with Digital Signatures

Currently the TupleBoard library supports one signature algorithm – RSADoubleSHA256Signature. Instances of the algorithm may be obtained through the provided algorithm factory class, *PKIAlgorithmFactory*. The factory automatically generates the necessary keys according to the bit strength specified. While any number of bits may be used, a value of 2048 is recommended as a minimum to achieve a reasonable level of security.

```
//Specify the bit strength of the algorithm to be created
int bitStrength = 2048;

//Obtain an instance of RSADoubleSHA256Signature
SignatureAlgorithm rsaSignature =
      PKIAlgorithmFactory
            .createEncryptionAlgorithm
                  (SignatureAlgorithms.RSAWithDoubleSHA256,
                   bitStrength,
                   helixRand);
```

Fig. 38 – Obtaining an instance of RSADoubleSHA256Signature

Once an instance of the signature algorithm has been obtained it may be used to compute signatures. It is often desired to sign an object, such as a tuple or a certificate. The following example demonstrates this signing process.

```
//Object to sign
Serializable toSign = new SomeObject(…);

//Sign object
byte[] signature = rsaSignature.signObject(toSign);

//Verify signature
boolean verified = rsaSignature.verifyObject(toSign, signature);

//Verify the operation was a success
if(verified) {
      System.out.println("Success");
}
```

Fig. 39 – Creating and Verifying a Digital Signature

Working with Certificates

In order to use the public key algorithms and signature algorithms shown in the examples above in a group setting there must be a way to distribute user algorithm data. In the TupleBoard library this is done through a certificate. The process begins by creating a certificate authority.

```
//Specify the CA's name.  This can be used as a group identifier
String caName = "myCA";

//Specify the file in which the CA will persist
File caFile = new File("myCA.ca");

//Create a CA Signature Algorithm for signing certificates
SignatureAlgorithm rsaSignature =
      PKIAlgorithmFactory
            .createEncryptionAlgorithm
                  (SignatureAlgorithms.RSAWithDoubleSHA256,
                   bitStrength,
                   helixRand);

//Create a public key algorithm to enable groupwide communication
PublicKeyAlgorithm rsa = PKIAlgorithmFactory
                        .createEncryptionAlgorithm
                              (EncryptionAlgorithms.RSA,
                               bitStrength,
                               helixRand);

//Create the new CA
CertificateAuthority c = new CertificateAuthority
                              (caName, groupEncryption, signer);

//Serialize the CA to disk for later retrieval
c.toStream(new FileOutputStream(caFile));
```

Fig. 40 – Creating a CA

With a CA defined it may now be used to generate a certificate for a user. This next example displays the creation of an *AlgorithmWallet* and an *AlgorithmWalletCertificate*, allowing the user to digitally sign messages and participate in secure communications.

```
//Specify user name
String userName = "user1";

//Specify start date of now
Calendar now = Calendar.getInstance();
Date startDate = now.getTime();

//Specify end date of a year from now
now.add(Calendar.YEAR, 1);
Date endDate = now.getTime();
```

```
//Specify file to store private wallet
File walletFile = new File("user1.wallet");

//Specify file to store certificate
File certFile = new File("user1.cert");

//Load the CA from disk
CertificateAuthority c = CertificateAuthority.fromStream
                              (new FileInputStream(caFile));

//Create the wallet and certificate
c.createAlgorithmWalletCertificate
      (startDate,
       endDate,
       username,
       certFile,
       walletFile);
```

Fig. 41 – Creating a Certificate

Working with Secure TupleBoard

The previous examples give insight into creation and use of some of the security primitives added to the API. To show how all of these features interact the final example demonstrates the design of a custom tuple, its signing, its encrypted transfer through the tuple board and its authentication. This example assumes two certificates sets have been generated as in the previous example, for users with id's user1 and user2. First, we define the sample tuple.

```
/**
 * This class represents a very simple extension of SignedTuple,
 * providing only a simple message field of type String.
 */
public class SecureExampleTuple extends SignedTuple {

      /**
       *Tuple payload - a simple String message
       */
      public final String message;


      /**
       * Template constructor.  This constructor is suitable for
       * creating default templates.  The message field is initialized
       * to null.
       */
      public SecureExampleTuple() {
            this.final = null;
      }
```

```
        /**
         * Tuple constructor.  This constructor is suitable for creating
         * a tuple with a message for posting.  Instances of this class
         * may not be created using this constructor.  Instead, use the
         * provided factory create method.
         */
        protected SecureExampleTuple(AlgorithmWalletCertificate cert,
                String message) {
                super(cert);
                this.message = message;
        }


        /**
         * Factory create method.  A new instance of this tuple is
         * created with the specified certificate and message.  The tuple
         * is signed using the specified algorithm before being returned.
         */
        public static SecureExampleTuple create
                (AlgorithmWalletCertificate cert,
                 String message, SignatureAlgorithm sigAlg)
                 throws IOException {

                //Create tuple
                SecureExampleTuple set = new SecureExampleTuple
                                                        (cert, message);

                //Sign tuple
                set.sign(sigAlg);

                //Return
                return set;
        }
}
```

With the tuple defined we may now write the code necessary for transferring one from user1 to user2. It is assumed that the both users' certificates are locally available for this example. First the posting code for user1:

```
//Read in my certificate
AlgorithmWalletCertificate myCert = AlgorithmWalletCertificate.
            fromStream(new FileInputStream(new File("user1.cert")));

//Read in my wallet
AlgorithmWallet myWallet = AlgorithmWallet.fromStream
                  (new FileInputStream(new File("user1.wallet")));

//Read in user2's certificate
AlgorithmWalletCertificate userTwoCert = AlgorithmWalletCertificate.
            fromStream(new FileInputStream(new File("user2.cert")));

//Create a new tuple to send
SecureExampleTuple t = SecureExampleTuple.create
                              (myCert, "Hello, user2!",
                               myWallet.getUserSignatureAlgorithm());
```

```
//Create the security context with which to send
//The security context's id is based on the target recipient's
//certificate
PublicKeySecurityContext pksc = new PublicKeySecurityContext
            (new CertificateSecurityContextId(userTwoCert),
                        userTwoCert.getUserEncryptionAlgorithm(),
                        helixRand);

//Create the tuple board
TupleBoard board = new TupleBoard();

//Post tuple and wait
board.post(t, pksc);
Thread.currentThread().join();
```

Now the reading code for user2:

```
//Read in my certificate
AlgorithmWalletCertificate myCert = AlgorithmWalletCertificate.
            fromStream(new FileInputStream(new File("user2.cert")));

//Read in user2's certificate
AlgorithmWalletCertificate userOneCert = AlgorithmWalletCertificate.
            fromStream(new FileInputStream(new File("user1.cert")));


//Create the security context with which to read
//The security context's id is based on my certificate
PublicKeySecurityContext pksc = new PublicKeySecurityContext
            (new CertificateSecurityContextId(myCert),
                        myCert.getUserEncryptionAlgorithm(),
                        helixRand);

//Create the tuple board
TupleBoard board = new TupleBoard();

//Read tuple
SecureExampleTuple received = board.read(new SecureExampleTuple());

//Verify signature and print message
if(received.isAuthentic(myCert.
                  getCASignatureVerificationAlgorithm())) {

      System.out.println("Message: " + received.message);
}
```

Fig. 42 – Working with Secure TupleBoard

Deliverables

The following is a list of all deliverables for this project:

- Required Libraries
  - jid3lib-0.5.4.jar – Java ID3 Tag Library
  - AbsoluteLayout.jar – Netbeans Swing Absolute Positioning Support
- Source Code
  - tbsrc.jar – Secure TupleBoard API
  - musicgroupsrc.jar – MusicGroup Application
- Executables
  - tb.jar – Secure TupleBoard API
  - musicgroup.jar – MusicGroupApplication
- Documentation
  - Secure TupleBoard JavaDoc
  - MusicGroup Application JavaDoc
  - Final Report (this document)
- Data
  - Certificate Authority
  - User Certificates
  - Performance Evaluation Script

All deliverables mentioned in this section as well as all other documents generated during the course of this project may be found on my project website at
http://www.cs.rit.edu/~krm4686/msproj/

Future Work

I am fairly happy with the results of this project but as with any project there is always additional work that can be done. The public key framework added to the TupleBoard library was designed to easily allow future extensions. The addition of additional algorithms to this framework would prove an interesting investigation. RSA is a very well known algorithm and has been in use for many years. Recently there has been interest in a new type of public key algorithm based on elliptic curves. Much like RSA, these algorithms can be used for encryption as well as digital signatures.

Similarly, it may be interesting to add additional symmetric key algorithms to the library. This would require a new layer of abstraction to allow security contexts to work with algorithms besides Helix, the only one currently supported. AES, the Advanced Encryption Standard, or Phelix, a new revised version of Helix may be good places to start.

Writing this final report has also given me some additional perspective on the project as a whole. There are a few things I would do differently if starting again. Currently authorization is only supported at the application level. It would be beneficial to integrate a role based authorization mechanism directly into the TupleBoard. I would also create a set of *SignedTupleListeners* which would extend the *PostListeners* and *PostWithdrawListeners* to allow for automatic authentication of read tuples before triggering posted events.

Both the application and the library were designed and developed for Java5. Upgrading to Java6 would provide some interface improvements and simplifications for the graphical client due to additions to the *javax.swing* package. Automatic table sorting and filtering as well as custom rendering on menu tab strips to enable close buttons on the private conversation tabs would be the first things that could be improved.

Finally, as with any trust based system there are some assumptions made regarding observing protocols and behaving in the expected manner. This illustrates the importance of a certificate authority only distributing certificates to users known to be good. While it is impossible to prevent all misuse, an easy improvement would be the inclusion of mp3 file hashing in the authorization certificates. This could prevent the modification of ID3 tag data in files to allow distribution of files masquerading as authorized files between users. But again this can do little to stop malicious activity between colluding users since it would be easy to sidestep the protocols in place.

I think the secure TupleBoard provides an excellent ad hoc collaborative application base and am interested to see what additional projects are done using it and extending it.

References

[1] Farng, Eric. Java ID3 Tag Library. 2006. http://javamusictag.sourceforge.net/index.html

[2] Ferguson, Niels and Schneier, Bruce. Practical Cryptography. Indianapolis: Wiley Publishing, Inc. 2003.

[3] Ferguson, Niels, Whiting, Doug, Schneier, Bruce, Kelsey, John, Lucks, Stephan and Kohno, Tadayoshi. *Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive*. In *Fast Software Encryption, 10th International Workshop, FSE'03*, volume 2887 of *Lecture Notes in Computer Science*, pages 330-346. Springer-Verlag, 2003.

[4] Gelernter, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.* 7, 1 (Jan. 1985), 80-112. DOI= http://doi.acm.org/10.1145/2363.2433

[5] IBM's TSpaces: Technology Overview. (Aug. 2003). http://www.alphaworks.ibm.com/tech/tspaces

[6] The Java Tutorials: *How to Use Tables*. http://java.sun.com/docs/books/tutorial/uiswing/components/table.html

[7] Kaminsky, Alan. TupleBoard Library. http://www.cs.rit.edu/~ark/tb.shtml

[8] Kim, Jisoo. *Group Key Agreement Protocols with Implicit Key Authentication*. RIT Computer Science M.S. project, July 2005. http://www.cs.rit.edu:8080/ms/static/spr/2004/4/jsk4445/index.html

[9] Kwon, Minseok. *Secure Group Communications*. RIT CS Colloquia Series. May 15, 2007. http://www.cs.rit.edu/~spr/CLQABS/jmk1.html

[10] Microsoft Cryptography API Reference. http://msdn2.microsoft.com/en-us/library/aa380256.aspx

[11] NetBeans IDE. http://www.netbeans.org/