UNIVERSITÀ DEGLI STUDI DI PISA

FACOLTÀ DI INGEGNERIA

LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

Tesi di Laurea Specialistica

PROGETTO E REALIZZAZIONE DI UN PROTOCOLLO SICURO PER IL CARICAMENTO REMOTO DI COMPONENTI SOFTWARE SU RETI DI SENSORI

Relatori:

Prof. Gianluca Dini

Prof. Francesco Marcelloni

Candidato:

Alfonsa Claudia Orlando

ANNO ACCADEMICO 2006–2007

Alla mia famiglia, ai miei amici e a tutte le persone che in questi anni mi sono state vicine e mi hanno dato un forte sostegno.

Sommario

Negli ultimi anni si é assistito allo sviluppo e la diffusione delle *Wireless Sensor Network*. Le Wireless Sensor Network sono costituite da dispositivi multifunzionali, a bassa potenza e dai costi contenuti, che possono comunicare fra loro tramite tecnologia wireless. Queste reti trovano molteplici applicazioni sia in campo militare,ambientale, industriale che domestico. In alcune applicazioni i dati, che i sensori si scambiano fra loro, possono essere sensibili per cui sono richieste le garanzie di sicurezza quali autenticitá, confidenzialitá ed integritá. La maggior parte dei protocolli di sicurezza correntemente utilizzati in Internet o nelle reti mobili, non possono essere utilizzati nelle Sensor Network a causa dei stringenti vincoli fisici. Ad esempio, non é possibile applicare algoritmi che si basano sulla crittografia a chiave pubblica, poiché richiedono molta memoria per immagazzinare i dati e sono computazionalmente costosi.Molte applicazioni per Wireless Sensor Network seguono il modello di comunicazione a gruppo in cui il numero e l'identitá dei nodi della rete variano dinamicamente nel tempo; questo introduce le richieste di garanzia di sicurezza di *Forward Security* e *Backward Security*. Se, ad esempio, ad un gruppo esistente si aggiunge un nodo, il server genera e distribuisce ai nodi la nuova chiave di comunicazione necessaria affinché il nodo non sia in grado di decifrare i messaggi inviati precendentemente l'istante di modifica del gruppo.
L'obiettivo della tesi verte sul problema del caricamento dinamico di componenti software all'interno di una Wireless Sensor Network. Le componenti software vengono inviate ai sensori in modo da garantire i principi di autenticazione e confidenzialitá. Queste componenti servono per potere programmare a distanza i sensori in seguito a possibili cambiamenti repentini della rete. Ad esempio, ipotizzando una situazione in cui si sia verificato un incidente in seguito al quale é stato distrutto l'intero cablaggio fisico della rete, é richiesto l'intervento dei pompieri

per potere fronteggiare la situazione di pericolo. Ogni pompiere viene rappresentato per mezzo di un Pc il quale comunica via wireless con i sensori. Ai sensori vengono distribuiti compiti diversi,in momenti distinti, per riportare la situazione in uno stato normale.I sensori devono garantire, con certezza, l'identificazione del Pc che ha inviato la componente.Per soddisfare il principio di autenticazione, a causa dei vincoli stringenti di memoria dei sensori (48 Kbytes) di flash), non é possibile usare la firma digitale in quanto comporta un grosso spreco di memoria. In alternativa é stato usato un meccanismo che si basa sulla costruzione di una catena di hash; dove la componente da inviare é suddivisa in pacchetti di dimensione fissa di 28 bytes. Partendo dall'ultimo pacchetto si calcola l'hash e si inserisce all'interno del pacchetto precedente. Questo processo iterativo é eseguito fin quando si arriva al primo pacchetto. Tale meccanismo garantisce che i sensori possono verificare l'autenticitá di tutti i pacchetti della componente e, nel caso in cui l'autenticazione di un pacchetto non é andata a buon fine, né viene interrotto il caricamento; inoltre questa procedura risponde a possibile attacchi di DOS. Affinché la chiave di sessione, utilizzata per la cifratura della componente software, venga inviata in modo sicuro ai sensori é stato usato il protocollo S^2RP che permette una distribuzione autentica delle chiavi. Questo protocollo oltre ad essere reattivo,scalabile ed efficiente,è ottimizzato per reti la cui topologia cambia frequentemente ed é basato su un modello di comunicazione a gruppo. Un prototipo del suddetto protocollo é stato implementato utilizzando come linguaggi di programmazione C e Java insieme alle loro middleware API; la piattaforma hardware é costituita da un nodo sensore *Tmote Sky* ed ha come sistema operativo Contiki.Per l'implementazione del lato server e del lato Pc sono state usate le librerie crittografiche di BouncyCastle.

Parte del progetto realizzato é stato utilizzato per un'applicazione appartenente allo sviluppo del progetto della Comunitá Europea *RUNES*.Lo scenario trattato riguarda l'interazione di diversi robot con differenti caratteristiche che si muovono in un ambiente condiviso senza supervisione.In particolare é stato fatto il porting in Java di un utility giá esistente chiamata *Codeprop*.Il lavoro di tesi svolto é stato organizzato in capitoli: i primi due introducono la descrizione delle *WSN* e del protocollo S^2RP mentre i capitoli successivi descrivono l'architettura del sistema. Il capitolo conclusivo tratta della verifica delle prestazioni del protocollo implementato.

Indice

Elenco delle figure

Capitolo 1

Wireless Sensors Network

Negli ultimi anni, grazie ai progressi tecnologici nei sistemi micro-elettro-meccanici,nelle comunicazioni wireless e nell'elettronica digitale si é assistito allo sviluppo di Nodi Sensori. Sono piccoli apparecchi a bassa potenza, dai costi contenuti, multifunzionali e capaci di comunicare tra loro tramite tecnologia wireless a raggio limitato.

I nodi sensori sono formati da componenti in grado di percepire dati dall'ambiente, in grado di elaborare dati e in grado di comunicare tra loro.

Una rete di sensori (Wireless Sensors Network) é costituita da un insieme di sensori disposti in prossimitá o all'interno del fenomeno da osservare.

La caratteristica dei nodi presenti all'interno di una rete é quella di cercare di comunicare tra loro. Sono costituiti da un processore on-board capace di ricevere dati, processarli e poi inviarli in una rete decentralizzata.

Le WSN possono essere usate in molte applicazioni come monitoraggio dell'ambiente e monitoraggio della salute. La realizzazione di queste applicazione richiede l'uso di diverse tecniche usate nelle reti wireless ad hoc. Molti degli algoritmi usati nelle reti ad-hoc non rispettano i requirement per le reti di sensori.

I principali motivi sono i seguenti:

1 Il numero di nodi che compongono una Rete di Sensori puó essere di alcuni ordini di grandezza maggiore del numero di nodi in una rete ad hoc.

2 Sono disposti con un alta densitá.

3 Sono soggetti a fallimenti.

4 La topologia cambia frequentemente.

5 Usano un paradigma di comunicazione broadcast, mentre la maggior parte delle reti ad hoc usano un paradigma point to point.

6 Hanno una capacitá limitata per quando riguarda l'alimentazione, il calcolo e la memoria.

7 Possono non avere un identificatore globale.

Ogni nodo sensore, presente all'interno delle rete puó ricevere dati da altri nodi presenti nella sua rete o in altra rete o da un server centralizzato oppure puó accumulare e instradare dati ad un utente finale.

I protocolli e gli algoritmi usati nelle Reti di Sensori devono avere capacitá auto-organizzative. I nodi presenti all'interno della rete non devono avere delle posizioni fisse ma devono essere in grado di potersi posizionare in posti difficilmente accessibili o in posti disastrati per i quali é necessario una disposizione random.

1.1 Caratteristiche di una rete wireless

Per la realizzazione e la progettazione di una Wireless Sensors Network bisogna tenere conto dei seguenti fattori:

1.1.1 Scalabilitá

La rete deve potere funzionare anche all'aumentare del numero di nodi. Si puó andare da pochi sensori a milioni di sensori.Per ottenere la scalabilitá si puó sfruttare la natura densa dei Sensori. La densitá dei nodi in una Rete di Sensori dipende dall'applicazione. Essa puó variare da pochi o da centinaia di nodi in una regione che puó essere meno di 10 metri di diametro.

1.1.2 Tollearanza ai guasti

La WSN deve essere resistente ai guasti. Deve potere continuare a funzionare quando alcuni sensori si rompono a causa di danni fisici, interferenze o battere scariche. Il protocollo e gli algoritmi devono potere garantire il livello di tolleranza richiesto dalla specifica applicazione per quella determinata rete.

1.1.3 Ambiente Operativo

I nodi sensori devono essere in grado di lavorare in qualsiasi tipo di ambiente.Essi sono disposti molto vicino o addirittura all'interno del fenomeno da osservare. Spesso si trovano a lavorare in zone geograficamente remote come in zone contaminate, o nei fondali di un oceano durante un tornado, o in un campo di battaglia o in un ambiente ad altissima temperatura. Quindi devono essere in grado di sopportare tutti gli ostacoli fisici, climatici e di pressione.

1.1.4 Topologia delle Rete di Sensori

I nodi possono essere disposti l'uno accanto all'altro all'interno di uno spazio geografico anche con una grande densitá.Questo richiede una grande attivitá per il mantenimento della topologia. Il mantenimento e il cambiamento della topologia puó essere diviso in tre fasi:

- *Pre-deployment e deployment phase.*I sensori sono disposti uno ad uno nell'ambiente o anche in blocco prima dell'avvio delle rilevazioni.

- *Post-deployment phase.*I cambiamenti topologici della rete sono dovuti al cambiamento della posizione dei nodi, all'energia disponibile o alla variazione della raggiungibilitá di un nodo.

- *Re-deployment of additional nodes phase.*Possono essere aggiunti in qualsiasi momento nodi sensori per sostituire nodi malfunzionanti oppure per far fronte alla dinamica dei task.L'aggiunta di nuovi nodi comporta la necessitá di riorganizzare la rete.Quindi sono necessari protocolli di routing specifici a causa della frequenza dei cambiamenti topologici e a causa dei vincoli imposti per il risparmio energetico.

1.1.5 Vincoli Hardware

Un nodo sensore é composto da quattro unitá base.

- Unitá di sensing: in genere é composta da 2 sottounitá: sensore e convertitore da analogico a digitale.

- Unitá computazionale: in genere é correlata ad una piccola unitá di immagazzinamento dei dati. Si occupa dei compiti specifici alla collabarazione del nodo con gli altri nodi della rete per portare a compimento il task assegnato.

- Unitá transceiver:permette la connessione del nodo alla rete. É possibile che sia o un apparecchio ottico o un apparecchio a radio frequenza o delle radio con basso duty-cycle. Quest'ultime, oggi, peró, danno grossi problemi.Viene consumata,infatti,molto energia per le operazioni di spegnimento.

- Unitá di energia: si puó ritenere,forse la componente piú importante dei nodi sensori. A volte puó essere supportata da una componente per il recupero dell'energia come per esempio una cella solare.

1.1.6 Mezzo di trasmissione

In una rete di sensori *multihop* i nodi possono interagire tra loro tramite un mezzo di comunicazione wireless. É quindi possibile utilizzare onde radio. Una possibilitá é quella di usare le bande ISM (industrial, scientific and medical).Un gruppo predefinito di bande che in molti casi sono license-free.

Oggi, la maggior parte dei sensori che sono in commercio, fanno uso di circuiti a Radio Frequenza. Un altra possibile tecnica per permettere la comunicazione dei nodi sensori é quella di usare gli infrarossi. La comunicazione con i raggi infrarossi ha i vantaggi di essere license-free e altamente resistente alle interferenze. I tranceiver che sfruttano gli infrarossi sono poco costosi e facili da usare. Il principale problema che si a nell'uso degli infrarossi é la necessitá di un interfacciamento diretto tra il Sender e il Receiver.Questo problema né impedisce l'uso in reti dove i nodi sono disposti in modo random.

1.1.7 Consumo Energetico

I sensori hanno una limitata sorgente di energia che in genere é minore di 0.5 Ah e 1.2V. Il suo cycle-life dipende, principalmente, dalla durata della batteria.

In una Rete di Sensori, i nodi possono sia ricevere dati che inviarli. La scomparsa di alcuni nodi puó portare a significativi cambiamenti topologici che possono richiedere una riorganizzazione della rete e delle politiche di routing. Per questa ragione molte ricerche oggi si stanno concentrando sulla realizzazione di alcuni protocolli e algoritmi *power-aware* ovvero protocolli che ottimizzano il consumo energetico.

Il consumo di energia dipende dai compiti svolti dal sensore quali:

- Sensing:la potenza necessaria per effettuare il campionamento si basa sulla natura dell'applicazione.

- Communication:la comunicazione comprende sia la ricezione che la trasmissione dei dati. Queste due attivitá hanno lo stesso consumo energetico.
- Data Processing: l'energia conusumata per eleaborare i dati. Questa energia é sicuramente molto piccola se comparata all'energia consumata per la comunicazione.

Delle tre attivitá la comunicazione ha il maggiore consumo energetico.

1.2 Sensori Tmote Sky

I Sensori *Tmote Sky* sono dei sistemi embedded ultra-power. Vengono usati in ambito industriale per creare reti di tipo mesh in grado di potere percepire informazioni ambientali come il livello di umiditá, la temperatura e la luce solare e fotovoltaica.

Possiedono un interfaccia che si basa sullo standard IEEE 802.15.4 che gli permette di potere interagire tra di loro.Le Tmote Sky hanno dei sensori integrati sulla scheda. Questi sensori integrati sono piú robusti rispetto a quelli esterni. In questo modo né viene ridotto il costo e anche la dimensione del package.

Le sue principali caratteristiche sono le seguenti:

- 250kbps 2.4GHz IEEE 802.15.4 Chipcon Wireless Transceiver
- Interoperabilitá con i dispositivi appartenenti allo standard IEEE 802.15.4
- Microcontrollare a 16-bit e 8MHz Texas Instruments MSP430(10K RAM,48k flash
- ADC,DAC,Supply Voltage Supervisor, e DMA Controller integrati
- Antenna integrada nella scheda con raggio di trasmissione di 50 m indoor e 125 m outdoor
- Sensori di umiditá temperatura e luce integrati
- Consumo Ultra Low current
- Fast wakeup from sleep (<6 μs)
- Programmazione e invio/ricezione dei dati tramite porta USB
- Supporto per le espansioni a 16-pin e optional connettore antenna SMA
- Supporto per TinyOs e Contiki: mesh networking e communication implementation

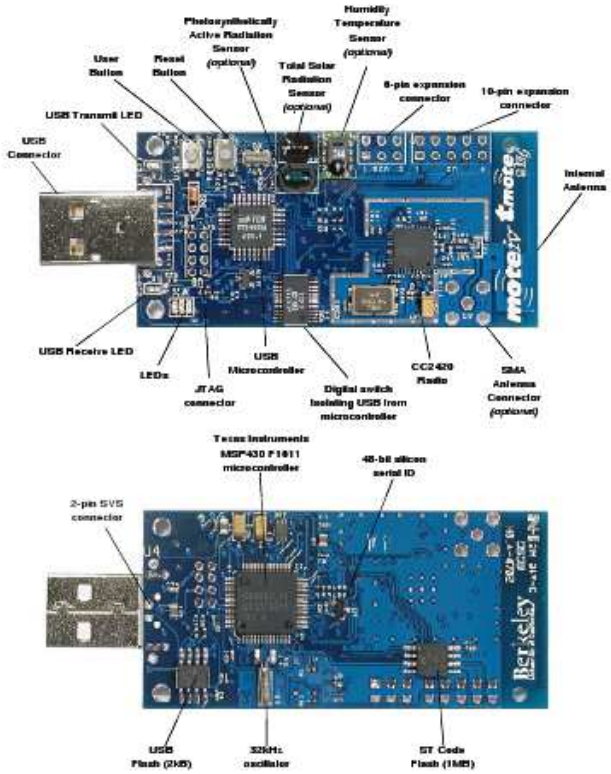


Figura 1.1: front and back di un module TSKY

1.2.1 Contiki

Contiki é il sistema operativo open-source sviluppato per le WSN.É stato progettato per sistemi embedded con vincoli di memoria. É altamente portabile e multi-tasking.

Ha un kernel event-driven in modo da gestire la concorrenza con poco consumo di memoria.Nel modello ad eventi i task non possono bloccarsi e non necessitano di fare polling, a differenza del modello a stack dove occorre riservare memoria per i contesti. Al di sopra del Kernel le applicazioni possono essere caricate dinamicamente a run-time.Contiki fá uso dei *protothreads* che permettono una stile di programmazione lineare e leggero.

Supporta il multi-threading preemptive e permette la comunicazione inter-processo attraverso lo scambio di messaggi in seguito all'arrivo di un evento.

Questo sistema operativo contiene due stack di comunicazione:uIP e Rime.uIP si basa sulla pila protocollare TCP-IP e quindi permette la comunicazione in rete.Rime é uno stack di comunicazione per raggi a breve distanza.Rime fornisce un vasto insieme di primitive di comunicazione, dalle comunicazioni best-effort e broadcast in un area locale al flooding dei dati multi-hop e affidabile.

Puó essere usato su diverse piattaforme da microcontrollori embedded come MSP430 e AVR a vecchi computer.

1.2.2 Contiki e Runes Middleware

Le WSN sono un esempio di sistema eterogeneo, distribuito e a larga scala. Sono costituite da un insieme di nodi che vogliono comunicare tra loro per la realizzazione di obiettivi comuni come fronteggiare una possibile situazione di pericolo o di emergenza. Gli sviluppatori per la realizzazione di questi sistemi complessi hanno bisogno di un particolare framework. L'architettura Runes mira a fornire questo particolare framework sotto forma di *middleware* basate su componenti. Con il termine *middleware* si intende un insieme di programmi informatici che fungono da intermediari tra diverse applicazioni. In questo caso fanno da intermediari tra i diversi sensori.

L'architettura Runes é caratterizzata da una struttura a livelli mostrata in fig 1.3:

Come é possibile vedere dalla figura al di sopra delle specifiche call system del SO si ha uno strato di software che fá da supporto per le applicazioni distribuite complesse permettendo la comunicazione e l'interazione dei diversi sensori. Questo strato di software non é un layer monolitico,ma é altamente modellabile e modificabile. All'interno di una stessa componente

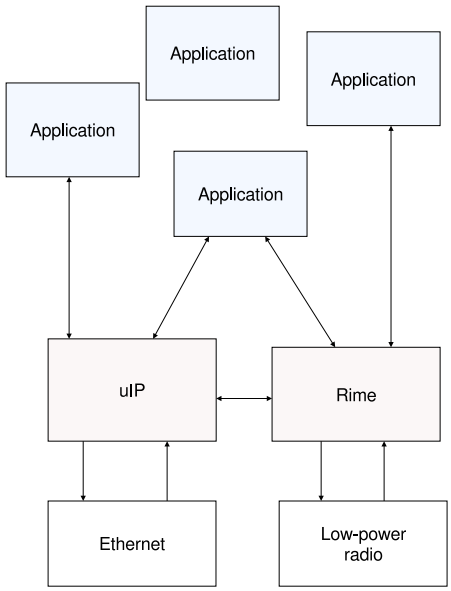


Figura 1.2: Contiki: 2 stack di comunicazione

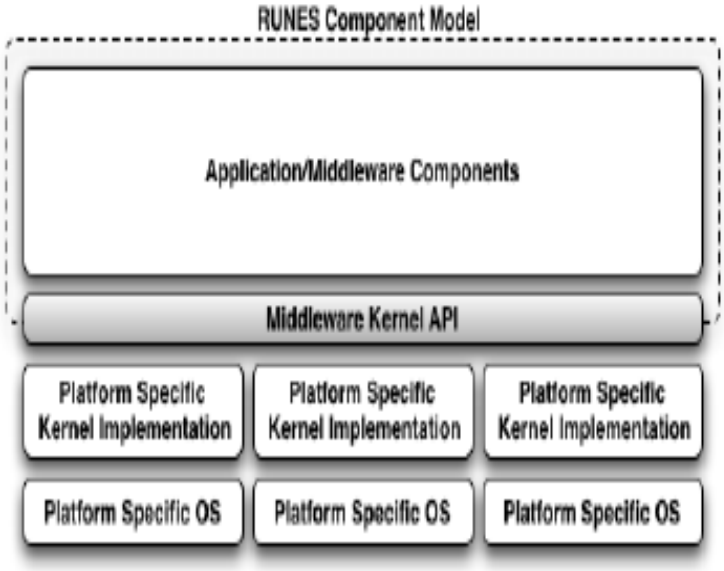


Figura 1.3: Architettura RUNES

potranno essere implementate diverse funzionalitá che possono essere usate separatamente e individualmente in base a dei vincoli sulle risorse o in base ai bisogni dell'applicazione. Per esempio all'interno di un modulo di comunicazione possono essere implementati protocolli best-effort e protocolli affidabili. Il middlewarekernel puó essere scritto con diversi linguaggi di programmazione come C o Java. Il layer superiore si occupa invece solo della definizione astratta delle diverse funzioni.

Il modello a componenti si basa sui seguenti elementi:

1. componente
2. component type
3. interfacce
4. recettacoli
5. connettori
6. connector factories

7. attributi

8. capsule

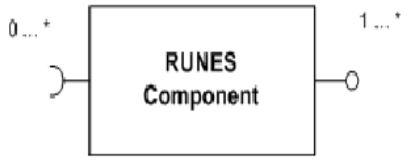


Figura 1.4: RUNES Component

Struttura di una componente

Una componente é un entitá che permette di realizzare specifici compiti.Permette ad alcune componenti di potere usare delle sue funzioni oppure puó chiamare funzioni definite da altre componenti. Offre servizi attraverso *interfacce* e usa servizi attraverso *recettacoli*. Puó essere instanziata a run-time da component type. In questo modo ogni type component puó essere usato per creare multiple istanze di componenti a run-time.

Un *interfaccia* é un insieme di funzioni che possono essere implementate da diverse componenti. A run-time puó essere scelto quale interfaccia usare.Rende visibile solo le dichiarazioni delle funzioni nascondendo quindi la loro implementazione.

Un *recettacolo* permette di potere usufruire dei servizi offerti dalle diverse componenti.Per potere collegare un recettacolo a un interfaccia si definisce un ulteriore componente chiamata connettore.

Questo modello definisce anche il concetto di Connectory Factory. Ci sono alcune componenti che,ogni qual volta viene invocata una chiamata al di sopra di una data connessione tra un recettacolo e un interfaccia, mandano in esecuzione determinate azioni. I connettori possono incapsulare arbitrarie funzionalitá che possono essere usate per monitorare o intercettare comunicazioni tra i corrispettivi recettacoli e interfacce. Ogni entitá (per esempio le componenti, component type, connettori, interfacce e recettacoli) é costituita dai seguenti elementi:

- *id* : identificativo dell'entitá
- *type*: tipo di entitá

- *attributes*: definiscono le caratteristiche di un entitá. Sono costituite da una coppia Key e Value e vengono contenute all'interno di un registro.

Una Capsula é un contenitore run-time che raggruppa tutte queste entitá.

1.2.3 Middleware Kernel API con java

Nel linguaggio Java i package che vengono usati per implementare le entitá del modello RUNES sono *runes.core* e *runes.core.capsuleInternals* che contengono le definizioni astratte delle entitiá e *runes.core.defaultImpl* e *runes.core. capsuleInternals. defaultImp* che contengono le implementazioni delle entitá.

Le interfacce *Java Component*, *ComponentFramework(Connectory Factory)* e *Connector* sono state implementate nelle classi astratte *BaseComponent*, *BaseComponentFramework* e *BaseConnector*. *BaseComponent* fornisce dei meccanismi che permettono di ritornare le interfacce o i recettaccoli implementati o definiti in una componente. *BaseComponentFramework* fornisce i meccanismi necessari per aggiungere e rimuovere vincoli su un *ComponentFramework*. *BaseConnector* implementa i meccanismi necessari per connettere un interfaccia a un connettore.

Per l'implementazione di default della capsula si usa la classe *DefaultCapsule*. Le classi *DefaultRegistry*, *DefaultBinder* e *DefaultLoader* sono usate per implementare funzionalitá specifiche delegate ai costituienti della capsula. Il *DefaultRegistry* é un registro che contiene tutti gli attributi delle entitá. e viene implementato attraverso una JavaHashMap. La classe *DefaultBinder* cerca di trovare il giusto Connettore basandosi sull'interfaccia da caricare e si occupa anche di istanziarlo. La classe *defaultLoader* si occupa del caricamento della *ComponentType* fornendogli come parametro il Pattern, che indica l'attuale percorso per trovare la componente da caricare. I dettagli per il caricamento di una componente sono presenti nelle classi *JavaComponentType* e *StringPattern*. La *JavaComponentType* implementa l'interfaccia *ComponentType* e si occupa di istanziare una nuova componente. La classe StringPattern si occupa di trovare un mapping tra la ComponentType cercata e il suo nome. *BaseComponent* fornisce,inoltre,i metodi per istanziare un oggetto recettacolo o un interfaccia.

1.2.4 Semplice applicazione

Si costruisca una componente chiamata *Calculator* che esegua le operazioni di addizione e di moltiplicazione.Queste operazioni saranno eseguite connettendo la componente *Cal-*

culator alle due componenti *Adder* e *Multipler*.L'architettura finale che si vuole ottenere é rappresentata nella figura 1.5.

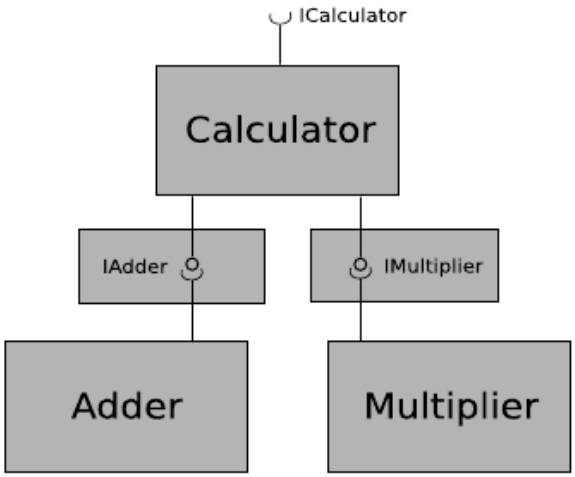


Figura 1.5: Architettura della componente Calculator

Si comincia costruendo le interfacce delle Componenti. In questo caso si costruiranno tre interfacce, una per la componente *Calculator* chiamata *ICalculator*, una per la componente *adder* chiamata IAdder e una per la componente *Multiplier* chiamata *IMultiplier*(Come mostrato nel listing 1.1).

```
public interface IAdder extends Interface{
public int add( int x,int y);
}

public interface IMultiplier extends Interface{
public int multiply( int x,int y);
}

public interface ICalculator extends Interface{
public int add( int x,int y);
public int multiply( int x,int y);
}
```

Listing 1.1: interfacce delle componenti *Adder*, *Multiplier*, *Calculator*

Dopo avere definito un interfaccia bisogna implentare la componente. Una semplice componente che implementa l'interfaccia *IAdder* é mostrata nel listing 1.2.L'addizione é eseguita

chiamando l'operazione aritmetica usuale di Java. Una volta che é definita un interfaccia lo sviluppatore puó decidere liberamente come implementarla. La componente deve estendere la classe BaseComponent per potere essere riconosciuta dal resto del framework. Lo sviluppatore deve implementare tutti i metodi definiti nell'interfaccia. Addizzionalmente deve inserire i metodi *construct()* e *destroy()* che offrono le funzionalitá di costruzione e distruzione della componente.

```
public class Adder extends BaseComponent implents Iadder{

public void construct()throws ComponentException{}

public void destroy()throws ComponentException{}

public int add(int x, int y){

return x+y;

}

}
```

Listing 1.2: implementazione componente *Adder*, *Multiplier*, *Calculator*

L'implementazione della componente *Calculator* é riportata nella figura 1.3. In questo caso il metodo *construct()* é responsabile della creazione di due *single receptacle* di cui la componente avrá bisogno per potersi connettere alle componenti *Adder* e *Multiplier*. I recettacoli verranno creati tramite il metodo *createSingleReceptacle(String InterfaceName)* e se né potrá recuperare una referenza con il metodo *getSingleReceptacleTo(StrinInterfaceName)*.Il recettacolo puó essere usato con il metodo *getSingleConnectedInterface()* che ritorna una referenza all'interfaccia connessa a recettacolo. Non c'é garanzia che un recettacolo sia connesso a un interfaccia. Per questa ragione il metodo *getSingleConnectedInterface()* deve essere invocato ogni volta che si usa un recettacolo.

Dopo avere creato le componenti bisogna istanziarle. Bisogna chiamare il metodo *load()* appartenenete alla classe *Capsule* che permette di caricare una componente e il metodo *instantiate()* che permette di instanziarlo. Nel listening 1.4 né é mostrato l'esempio di caricamento della componente *Calculator*.

A questo punto bisogna collegare le componenti tra loro. Si ipotizzi di volere collegare *Adder* a *Calculator* (come mostrato nel listening 1.5).Si recupera una referenza dell'interfaccia della componente *Adder* e una referenza del recettacolo della componente *Calculator*.Dopo

```
public class Calculator extends BaseComponent implements ICalculator{

public void construct()throws ComponentException{

createSingleReceptacle("runes.Iadder");
createSingleReceptacle("runes.IMultiplier");

}

public void destroy()throws ComponentException{}

public int add(int x, int y){

SingleReceptacle r=getSingleReceptacleTo("runes.Iadder")
return (IAdder) r.getSingleConnectedInterface().add(x,y);

}

public int multiply(int x, int y){

SingleReceptacle r=getSingleReceptacleTo("runes.IMultiplier")
return (IAdder) r.getSingleConnectedInterface().multiply(x,y);

}

}
```

Listing 1.3: implementazione componente *Calculator*, *Multiplier*, *Calculator*

```
ComponentType CompoType=Capsule.load(new StringPattern("runes.Calculator"));
Component Compo=Capsule.instantiate(CompoType);
```

Listing 1.4: caricamento componente *Calculator*, *Multiplier*, *Calculator*

avere recuperato queste referenze si invoca il metodo *bind()* della classe Capsule che ritorna un oggetto di tipo *Connector* che né rappresenta il legame.

```
Interface Adderif=(Interface)Capsule.getAttr(Compo,"INTERFACE–Runes.Iadder")
Receptacle Rec=(Receptacle)Capsule.getAttr(Compo,"RECEPTACLE–RUNES.
ICalculator")
Connector CalctoAdder=Capsule.bind(interfaceif,Rec)
```

Listing 1.5: bind tra *Calculator* e *Adder*

Capitolo 2

S^2RP

S^2RP é un esempio di protocollo sicuro e affidabile per una Wireless Sensor Network. Mira a creare un trade-off tra sicurezza e consumo di risorse di calcolo, di memoria e di energia. Garantisce una distribuzione autentica delle chiavi e soddisfa la *Backward Security* e la *Farward Security*.

I nodi sensori comunicano basandosi su un modello di comunicazione a gruppo. Un gruppo é costituito da un insieme di nodi sensori che vogliono scambiarsi informazioni in modo confidenziale e sicuro. I membri del gruppo condividono una chiave di gruppo con la quale cifrano i dati per evitare che possibili intrusi possano venire a conoscenza delle informazioni che si scambiano. Quando un nodo vuole far parte del gruppo, vengono informati tutti i nodi che né fanno parte e viene stabilita una chiave di gruppo tramite la quale tutti i membri potranno comunicare tra loro. Quando un nodo sensore lascia il gruppo vengono informati tutti i membri che né fanno parte e viene cambiata la chiave di gruppo con la quale poi comunicheranno.

2.1 Garanzie di Sicurezza

Il mezzo usato per la comunicazione é wireless. Quindi é necessario proteggere la comunicazione per evitare che un avversario possa intercettarla. Puó, infatti, posizionarsi all'interno del raggio di copertura con un ricetrasmettitore e poiché il mezzo é per sua natura broadcast puó facilmente intercettare i messaggi.

É necessario che siano garantiti i principi tradizionali della crittografia:

1 *Confidenzialitá*: I dati trasmessi devono essere letti solo dagli agenti autorizzati.

2 *Integritá*: Gli agenti devono stabilire con certezza l'identitá del nodo che ha creato i dati.

3 *Autenticitá* Gli agenti devono rilevare se i dati sono stati modificati prima di essere ricevuti.

Nelle reti tradizionali la sicurezza viene soddisfatta usando meccanismi end-to-end come IPSec,SSL,SSH. Nelle reti WSN non é possibile usare questo tipo di meccanismi poiché il traffico é one-to-many. Bisogna, inoltre usare dei meccanismi che non comportino un grosso spreco per quanto riguarda la potenza di calcolo e il consumo di energia.

La crittografia a chiave pubblica non puó essere usata perché richiede elevata potenza di calcolo e notevole quantitá di memoria. Per esempio l'algoritmo di cifratura RSA usa chiavi lunghe 1024 bit.Quindi verrá usata la crittogafia a chiave simmetrica. La stessa chiave condivisa verrá usata da tutti i membri del gruppo. Il gruppo varia dinamicamente e i diversi membri possono facilmente uscire ed entare quando meglio lo desiderano. Sono, infatti, presenti degli algoritmi di scambio delle chiavi abbastanza efficienti e semplici. Dato che siamo in presenza di un modello di comunicazione a gruppo bisognerá anche garantire:

1 *Forward Security*: Dopo che un agente ha lasciato il gruppo non deve potere piú decifrare le comunicazioni del gruppo.

Quando un nodo esce dal raggio di copertura considerato sicuro puó essere catturato da un avversario che puó intercettare tutte le comunicazioni che verranno fatte successivamente.

2 *Backward Security*: Non deve essere possibile per un agente che entra a fare parte del gruppo potere decifrare i messaggi precedentemente inviati.

Un avversario malizioso potrebbe, infatti, entrare a fare parte del gruppo per potersi impossessare dei messaggi precedentemente inviati.

Per potere garantire la *Forward Security* e la *Backward Security* il protocollo S^2RP utilizza due tecniche principali:

- *key-chain* É un meccanismo di autenticazione basato sullo schema di Lamport *One-time password* che permette di garantire l'autenticitá delle chiavi ricevute dai nodi sensori.
- *eviction-tree*: É una struttura dati che permette di limitare il numero di messaggi per gestire un *rekeying*.

2.1.1 Funzione hash

Per potere garantire l'integritá dei dati é possibile usare algoritmi di hashing. Tali algoritmi permettono di creare, data una stringa iniziale di lunghezza variabile, una sequenza di byte di lunghezza fissa chiamata *digest*. Poiché una funzione *hash* deve produrre un valore che puó essere considerato come una rappresentazione compatta dell'input, analiticamente sará una funzione di tipo molti a uno, cioé é possibile che due messaggi differenti abbiano lo stesso valore di hash. Affinché una funzione *hash* sia utilizzibile nei protocolli di sicurezza deve godere delle seguenti proprietá:

- *Compressione*: Dato un input di lunghezza arbitaria la funzione hash deve poterlo mappare in un output di lunghezza fissa.
- *Facilitá di calcolo e monodirezionalitá*: Deve essere computazionalmente facile dato un input *X* ottenere un *Y* tale che $y = h(x)$ e deve essere computazionalmente difficile effettuare l'operazione inversa.
- *Limitazioni delle collisioni*

Teoricamente essendo una funzione molti a uno, é possibile che due messaggi diversi diano luogo allo stesso valore *hash*, anche se ció deve verificarsi sporadicamente. In particolare é importante non solo che , dato X ,sia computazionalmente difficile trovare un X^1 tale che $h(X) = h(X^1)$, ma anche che dati due messaggi casuali X e X^1 sia difficile che $h(X) = h(X^1)$. La differenza tra i due casi é che nel primo caso é conosciuto X, mentre nel secondo caso i due messaggi sono casuali.

Proprio per questo é possibile distinguere tra funzioni hash monodirezionale (OWHF,One Way Hash Function), per cui vale la prima asserzione, e funzioni *hash* resistenti alla collisione (CRHF, Collision Resistance Hash Function) per cui vale la seconda asserzione (e quindi anche la prima). Nell'utilizzare gli algoritmi di hashing, si puó prevedre o meno l'utilizzo di una chiave, cioé l'algoritmo puó ricevere in ingresso, oltre al messaggio da comprimere, anche una chiave come parametro e quindi il digest dipende dalla chiave utilizzata. In questo caso il digest dipende anche dalla chiave e viene indicato con il nome MAC (Message Authentication Code), mentre nel caso in cui non dipende dalla chiave viene chiamato MDC (Message Detection Code).

2.1.2 Key Chain: un meccanismo per l'autenticazione

Il Key Chain é un meccanismo molto semplice che permette al ricevitore di potere verificare l'autenticitá dei dati ricevuti.

Esso si basa sul meccanismo *One Time Password*.

Si consideri il caso in cui un nodo Sender invia una chiave a un Receiver. Il Sender costruisce una catena di chiavi. Sceglie un seme iniziale *s* e costruisce la seguente catena:

s

K^N=*H(s)*

K^{N-1}=$H^1(s)$

K^{N-2}=$H^2(s)$

K^{N-3}=$H^3(s)$

K^{N-4}=$H^4(s)$

....

K^1=$H^{N-1}(s)$

K^0=$H^N(s)$ dove $H^i(s)$ corrisponde ad applicare i volte la funzione hash OWHF.

La catene ottenuta é definita come segue:

$$\boxed{Chain(K^N, N, H) = \{\forall i : 0 \leq i \leq N | K^i = H^{N-i}(K^N)\}}$$

Le proprietá delle funzioni hash garantiscono che chi conosce K^j puó calcolarsi tutte le chiavi K^i con $0 \leq i \leq j$, ma non puó calcolarsi le chiavi K^x con $j + 1 \leq x \leq N$. Il Sender rileva le chiavi della catena nell'ordine inverso rispetto a come sono state create. Comincerá, quindi, trasmettendo $H^N(s)$. Le potrá trasmettere anche in un canale insicuro.Il Receiver per verificarne l'autenticitá basta che applichi j-1 volte la funzione hash a K^0. Se sono uguali né viene verificata l'autenticitá. L'unico vincolo é che il Sender deve in qualche modo inviare K^0 al Receiver, in modo che questo possa autenticare tutte le chiavi del *key-chain*.

2.1.3 Eviction-Tree

La struttura dati fondamentale per l'implentazione di questo protocollo é l'*eviction-tree*. É un albero binario che definisce una struttura gerarchica delle chiavi.

Ad ogni nodo interno é associata una catena Chain(K^N,N,H). Ad ogni foglia dell'albero che corrisponde a un membro del gruppo, é associata una chiave privata. Questa é una chiave

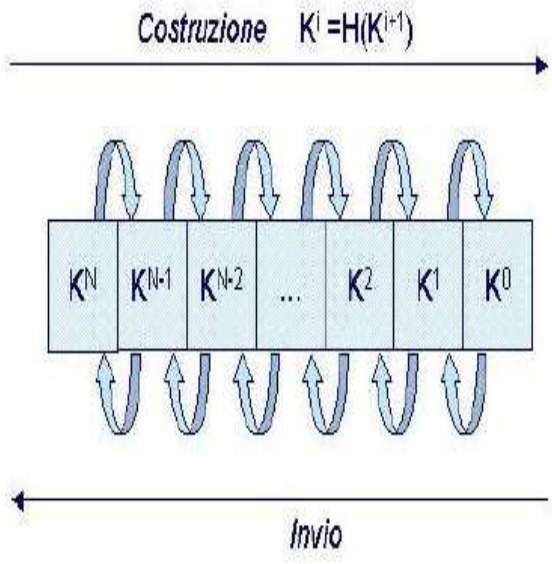


Figura 2.1: Meccanismo del KeyChain

simmetrica che condivide con il *KDS* (*KeyDistributionService*). La gestione completa dell'albero,e quindi del protocollo, viene eseguita dal gestore del gruppo, mentre i nodi sensori non conoscono l'albero.

Dato un nodo sensore s_x si definisce *Path(s_x) l'insieme dei nodi interni facenti parte del percorso che vá dalla radice dell'albero alla foglia associata al sensore s_x*.

Si definisce *KeySet(s_x)* l'insieme delle chiavi correnti dei nodi interni appartenenti al *Path(s_x)*.

$$KeySet(s_x) = \{K_{e_j}^{cur} | e_j \in Path(s_x)\}$$

Il KeySet relativo ad ogni foglia é composto da *h* chiavi con *h* pari al numero di livelli dell'eviction-tree.Il numero di livelli dell'albero é pari a *log(n)* con *n* numero di foglie dell'albero.

Nel gruppo possono essere formati dei sottoinsiemi o cluster dove i nodi che né fanno parte sono tutti discendenti di uno stesso nodo padre diverso dal nodo radice. Considerando i sottoinsiemi cosí definiti si chiama $K_{cluster}$ la chiave condivisa da tutti i membri facenti parte del medesimo sottoinsieme.

Questa struttura permette di garantire la *forward security*. Quando un nodo sensore ab-

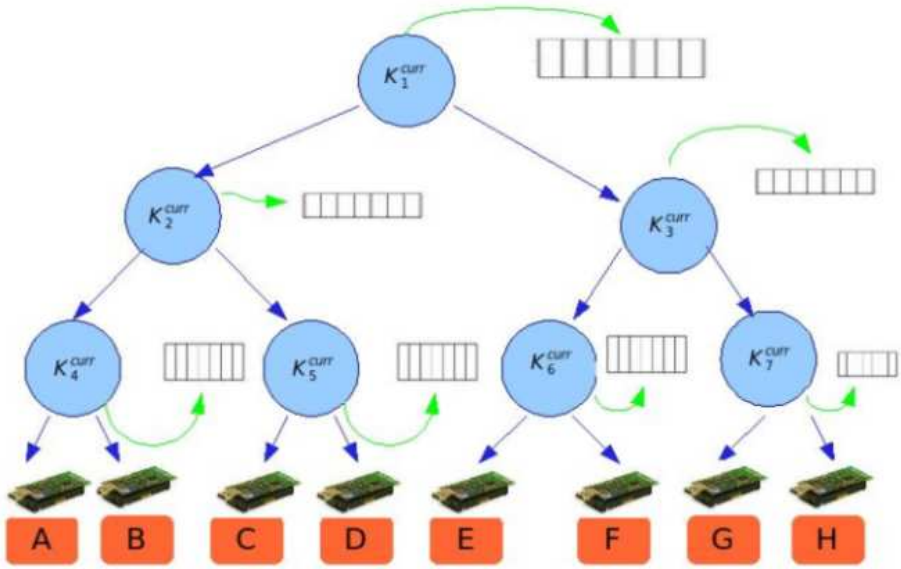


Figura 2.2: Struttura generale dell'Eviction-Tree

bandona la comunicazione tutte le chiavi nel KeySet associato al medesimo sensore sono compromesse e devono essere sostituite. Il *KDS* aggiorna le chiavi correnti come segue: $\forall e_j \in Path(sensore)$ il KDS rimpiazza la $K_{e_j}^{cur}$ con la $K_{e_j}^{next}$ del Chain($K_{e_j}^{N}$,N,H). Il *KDS* trasmette la chiave aggiornata a tutti i discendenti del nodo interno che ha compromesso la sua chiave. Naturalmente questa chiave non verrá inviata al nodo che ha voluto abbandonare la comunicazione.

L'Evinction-Tree non garantisce, peró la Backward Security. Quando un nodo sensore, infatti, si unisce al gruppo , riceve le chiavi correnti dei nodi interni appartenenti al suo percorso. Si puó calcolare, quindi tutte le precedenti chiavi dei nodi interni appartenenti al suo percorso. Per risolvere questo problema, il KDS definisce un ulteriore chiave K_j chiamata *chiave di join*. Questa chiave é condivisa da tutti i membri del gruppo ed é aggiornata quando arriva un nuovo membro nel gruppo. Serve per generare la nuova chiave di gruppo:

$$\boxed{Key_{grp} = M(Key_{join}, K_{e_0})}$$

Per limitare il numero di messaggi necessari per comunicare ad ogni sensore la K_J corrente, questa viene comunicata a s_x quando di verifica un evento di aggiunta di un elemento al gruppo. Le K_J successive poi vengono calcolate in modo distribuito come digest del K_J precedente.Il gestore di gruppo comunica ai membri del gruppo di calcolarsi la nuova K_J inviando loro un comando che chiameremo K_C.

2.2 Architettura del Sistema

L'archiettura del gestore del gruppo *Group Controller* (GC) é composta da 3 moduli principali:

1 *Group Membership Service* (GMS): gestisce le operazioni del gruppo. Se un nodo volesse entrare a fare parte del gruppo dovrebbe fare richiesta al KDS (Key Distribution Service). Il GMS genera i comandi per il rinnovo delle chiavi e si occupa di informare il KDS, che é il vero e proprio gestore del protocollo.

2 *Key Distribution Service* (KDS):É il fulcro del protocollo.Gestisce il protocollo, conosce l'*eviction-tree*, la catena delle chiavi dei nodi interni e quella delle chiavi di comando.Questo modulo riceve le segnalazioni degli eventi di rekeying dal *GMS* e gestisce le fasi principali del protocollo.

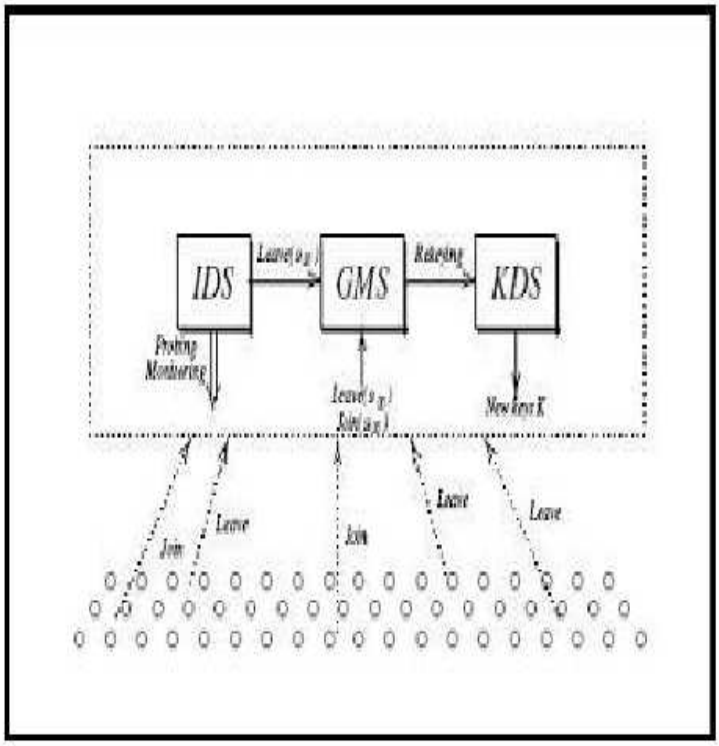


Figura 2.3: Architettura del Sistema

3 *Intrusion Detection System* (IDS): si occupa di verificare se ci sono situazioni di intrusioni da parte di avversari.Le WSN sono caratterizzate da una limitata sorveglianza sui nodi. Quindi risulta possibile che un sensore sia sottratto, riprogrammato con funzionalitá pericolose per il gruppo e reintrodotto nella rete. L'IDS monitora la rete e se si verifica una situazione di manomissione forza l'uscita dal gruppo dei sensori compromessi.

2.3 Fasi del protocollo

Le fasi del protocollo sono principalmente due:

- fase di inizializzazione
- comunicazioni sicure tra i membri del gruppo

2.3.1 fase di inizializzazione

In questa fase il KDS inizializza tutti i nodi sensori facenti parte dell'albero. Avviene in modo off-line.

Il KDS assegna ad ogni nodo sensore un identificatore e una chiave privata. Per calcolare la chiave privata di ogni nodo sensore si esegue la seguente operazione:

$\boxed{K_{sa} = f(MK, sa)}$

dove

sa=identificatore sensore

f=funzione sicura pseudo-random di Mixing

MK=MasterKey

In questo modo il KDS deve solo memorizzare la MasterKey, senza occupare memoria per tenere traccia di tutte le chiavi dei sensori dell'albero. Quando deve comunicare con uno specifico sensore si calcolerá la sua chiave privata corrispondente.

Il KDS calcola il Key-Chian di ogni nodo interno dell'albero e invia ad ogni membro del gruppo, in modo sicuro:

- le teste delle catene associate ai nodi interni dei loro specifici percorsi
- la chiave di join
- la chiave di Command

2.3.2 Comunicazioni sicure di gruppo

Questa é la fase durante la quale tutti i nodi del gruppo possono scambiarsi dati e comunicare con il server. Il KDS garantisce,sempre, la Backward Security e la Forward Security per mezzo dell'operazione di Rekeying. Periodicamente cambia la chiave di gruppo e la chiave di cluster in modo da limitare la quantitá di dati che gli avversari possono memorizzare.

Ogni membro del gruppo tiene, sempre, in memoria la chiave di join e il suo KeySet. Queste chiavi vengono usate per generare localmente le chiavi di gruppo e le chiavi di cluster.

La chiave di gruppo Key_{grp} viene ottenuta nel seguente modo:

$$\boxed{Key_{grp} = M(Key_{join}, K_{e_0})}$$

dove

M=funzione di Mixing

K_{e_0}= key chain della radice dell'albero

La chiave di cluster $Key_{cluster}$ viene ottenuta, invece, nel seguente modo:

$$\boxed{Key_{cluster} = M(Key_{join}, K_{e_i}^{cur})}$$

dove

M=funzione di Mixing

$K_{e_i}^{cur}$= key-chain corrente associato al nodo padre dell'i-esimo cluster

Le operazioni che é possibile eseguire sono le seguenti:

1. Evento Left
2. Evento di Join

2.3.3 Evento Left

Quando un sensore fá esplicita richiesta al KDS di volersi allontanare dal gruppo o é costretto a lasciarlo a causa di un fenomeno di intrusione si verifica un evento di left. La chiave che il nodo sensore possiede é considerata compromessa e il KDS deve rinovvare le chiavi appartenenti al KeySet del nodo sensore.

Il KDS identifica i nodi sensori compromessi che appartengono al Path del sensore.Aggiorna l'Eviction-Tree eliminando la foglia associata al sensore stesso e recupera le chiavi successivi dei nodi interni e le invia ai nodi figli e allo stesso sensore.

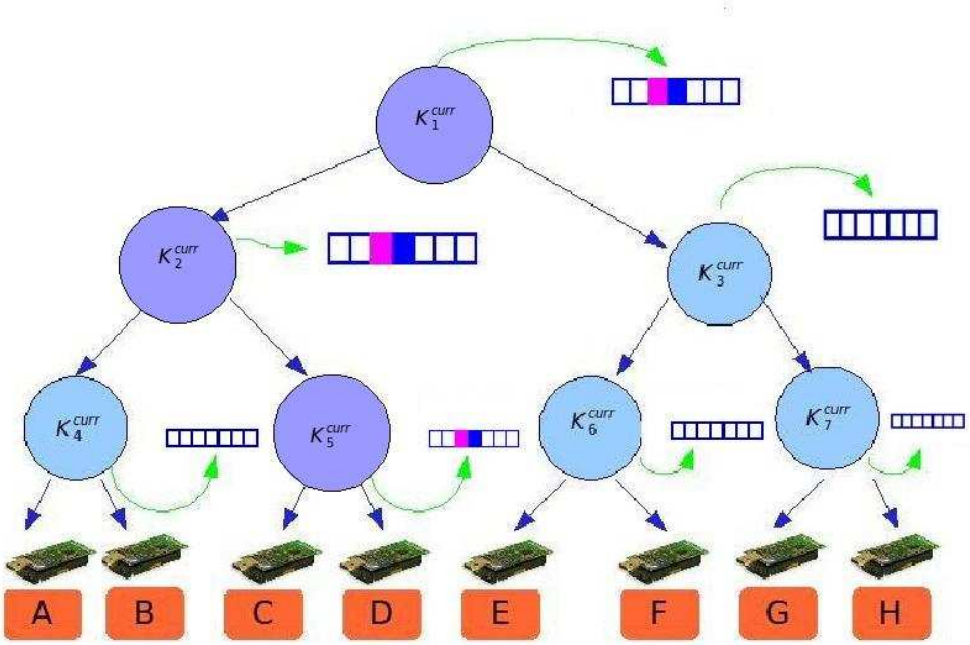


Figura 2.4: Esempio di evento left

Consideriamo il caso in cui il nodo D lascia il gruppo. Sia K_{sC} la chiave privata associata al sensore che ha come identificativo *C*. Per limitare il numero di messaggi il server li invia nel seguente ordine:

M1: $KeyServer \rightarrow C$: $E_{K_{sC}}(K_{e_5}^{curr})$
M2: $KeyServer \rightarrow C$: $E_{K_{e_5^{curr}}}(K_{e_2}^{curr})$
M3: $KeyServer \rightarrow A, B$: $E_{K_{e_2^{curr}}}(K_{e_4}^{curr})$
M4: $KeyServer \rightarrow A, B, C$: $E_{K_{e_2^{curr}}}(K_{e_1}^{curr})$
M5: $KeyServer \rightarrow E, F, G, H$: $E_{K_{e_3^{curr}}}(K_{e_1}^{curr})$

Nel caso riportato in figura il nodo C quando riceve i messaggi M1,M2 e M4 esegue le seguenti azioni:

- Decifra M1 con la propria chiave privata K_sC
- Verifica che $K_{e_5^{curr}}$=$H(K_{e_5^{next}})$
- Decifra M2 con $K_{e_5^{next}}$
- Verifica che $K_{e_2^{curr}}$=$H(K_{e_2^{next}})$
- Decifra M3 con $K_{e_2^{next}}$
- Verifica che $K_{e_4^{curr}}$=$H(K_{e_4^{next}})$
- Decifra M4 con $K_{e_2^{next}}$
- Verifica che $K_{e_1^{curr}}$=$H(K_{e_1^{next}})$

2.3.4 Evento di Join

L'evento di Join si verifica quando un nuovo nodo vuole entrare a fare parte del gruppo. Per garantire la Backward Security il KDS aggiorna la chiave di join K_J. Quando un nodo fá esplicita richiesta di entrare nel gruppo il KDS invia in broadcast il comando K_J. Ipotizziamo che il sensore H voglia entrare nel gruppo e che tutti i membri del gruppo abbiano immagazzinato K_C^l.

Il Messaggio inviato dal KDS trasporta il comando K_C^{l+1}.

I membri del gruppo né verificano l'autenticitá vedendo se (K_C^{cur}= K_C^l) == H(K_C^{next}=K_C^{l+1}).

Se il controllo ha successo ogni membro del gruppo calcola localmente la nuova chiave di join semplicemente facendo K_J=$H_J(K_J)$.

A questo punto i membri del gruppo rinfrescano la loro chiave di comando.

Il KDS si occupa anche di inviare in modo unicast al sensore che vuole entrare nel gruppo K_C^{next} criptata con la sua chiave privata.

2.4 Refreshing delle chiavi

Periodicamente il KMS rinfresca la chiave di cluster e la chiave di gruppo.In questo protocollo queste due chiavi sono funzioni della chiave di join. Di conseguenza il KDS si preoccupa dopo ogni intervallo di tempo di inviare in broadcast K_C^{next}. Quando ogni nodo riceve questo messaggio né verifica l'autenticitá e si calcola la nuova chiave di join aggiornando cosí le due precedenti chiavi.

2.5 Riconfigurazioni delle chiavi

I Key-Chain associati ad ogni nodo interno e quello associato alla catena dei comandi hanno una lunghezza limitata. Quando tutte le chiavi appartenenti a un Key-Chain sono state usate il Key-chain viene riconfigurato dal KDS.

Il KDS si comporta diversamente se deve riconfigurare il Key-Chain associato alla catena di comando oppure quello associato ad ogni nodo interno. Nel primo caso invia in modo unicast ad ogni nodo sensore del gruppo la testa della catena K_C. Nel secondo caso. invece, invia in modo unicast K_{e_j} a tutti i nodi sensori percorsi dal nodo interno e_j

Capitolo 3

Scenario Base

Il protocollo S^2RP puó essere usato per creare un ulteriore livello che si occupa del caricamento remoto di *Componenti* in maniera autentica e confidenziale all'interno di una *WSN*.Una *WSN* é composta da una serie di dispositvi mobili dotati di sensori che supportano le elaborazioni e che consentono le comunicazioni con gli altri dispositivi. Si ipotizzi, per esempio, di disporre di una *WSN* per potere monitorare gli equipaggiamenti e le munizioni di un esercito. Vi saranno una serie di dispositivi mobili sparsi per il raggio di copertura della *WSN* che si occupano di controllare il numero di munizioni e altri che controllano che nessuna persona non autorizzata entri all'interno del raggio. Ogni sensore dovrá essere programmato per potere eseguire i propri compiti. All'interno della rete vi saranno dei *PC* che si occuperanno di programmare i vari sensori a distanza.Per permettere il tunneling sicuro e affidabile dei dati sensibili il Pc e il nodo sensore dovranno condividere una chiave di sessione. Questa chiave verrá generata e usata dal PC per cifrare la *componente* e dal dispositivo mobile per decifrarla.Per potere permettere l'accesso solo ai Pc autorizzati vi sará una terza entitá chiamata *Server* che si occuperá di controllare le credenzialitá dei vari Pc e in base alle sue credenzialitá renderá possibile il caricamento. Il Pc invierá questa chiave al sensore, il quale verificherá che la chiave é autentica.

In seguito a cambiamenti ambientali (per esempio intruso nel dispositivo di munizioni) lo stato della rete puó modificarsi per fronteggiare la nuova situazione. In relazione a questi cambiamenti alcuni PC potrebbero non avere piú il diritto di caricare componenti nella *WSN*. Per simulare il cambiamento di stato della rete il *Server* imposta un Timer che periodicamente scatta. Quando scatta viene modificato lo stato associato al *Server* che puó essere normale o

di emergenza. Quando cambia lo stato il *Server* aggiorna l'eviction-tree e manda una serie di pacchetti per aggiornare i sensori.

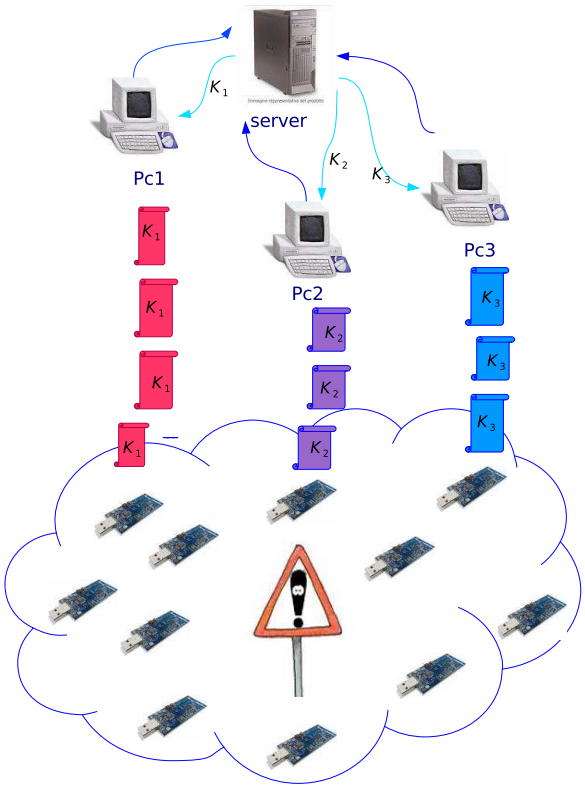


Figura 3.1: Scenario Base

Lo scenario é basato sui seguenti attori:

- Una sorgente (per esempio un PC) che vuole inviare del codice (nel nostro caso sotto-forma di componenti RUNES) a una mote.
- Un server che interagisce con i Pc per la creazione di una specifica chiave di sessione simmetrica. Questa chiave verrá inviata alla mote su cui intende caricare la componente. La componente verrá cifrata con questa chiave di sessione in modo da poterne garantire la confidenzialitá e l'autenticitá.
- La mote che riceve la chiave di sessione e la componente. Né verifica l'autenticitá. Se la componente é autentica, allora, la decifra e la carica.

3.1 Lato Server

Il lato Server é il fulcro del protocollo.Interagisce con i Pc e con le mote per permettere di stabilire una comunicazione sicura tra loro.

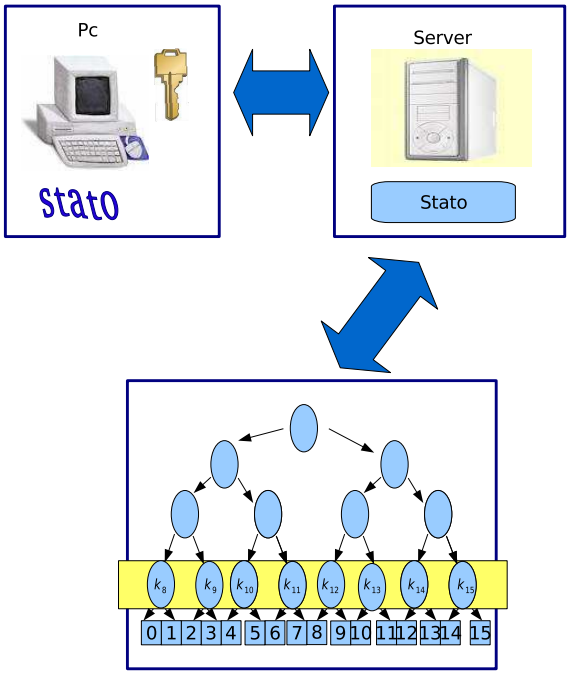


Figura 3.2: Interazione del Server con i Pc e le Mote

É caratterizzato da:

1. una chiave pubblica e privata
2. uno stato che puó essere Normale o di Emergenza.

É costituito principalmente da due moduli *il KM* (Key Mangament) e il *IPC* (Interaction PC).

Il *KM* interagisce con il *KDS* (Key Distribution Server). Il *KDS* é una componente del gestore di gruppo del protocollo S^2RP (analizzato nel capitolo 2) che permettere la creazione,

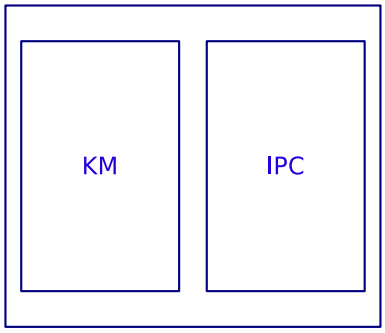


Figura 3.3: Struttura del Server

l'inizializzazione e la modifica dell'eviction-Tree (albero binario che definisce una struttuta gerarchica delle chiavi dei sensori del protocollo S^2RP (paragrafo 2.1.3)) a seguito di eventi di refresh,join e left.

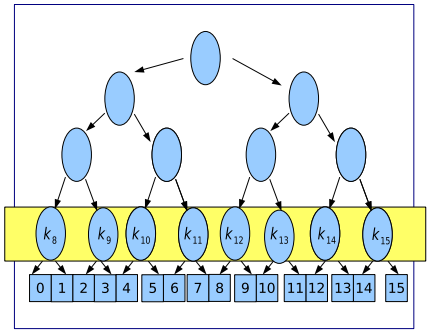


Figura 3.4: Rappresentazione dell'albero

L'eviction-tree (come si puó vedere dalla figura 3.4) ha un numero di livelli pari a 4 e un arietá uguale a 2.

Il KM (come é possibile vedere dalla figura 3.5), per potere simulare il comportamento di un dispositivo esterno che cambia lo stato della rete in relazione a dei fattori esterni, attiva

un timer che periodicamente modifica il suo stato da Normale a Emergenza o da Emergenza a Normale.

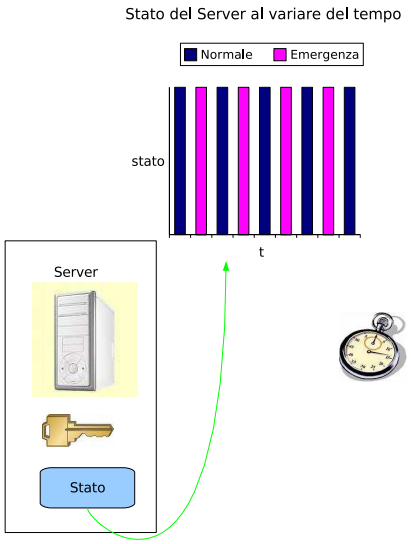


Figura 3.5: Modifica dello stato del Server al variare del tempo

Al variare dello suo stato si occuperá di aggiornare l'eviction-tree e di inviare a tutti i nodi sensori dell'albero la chiave di cluster aggiornata del nodo padre.

Si consideri l'esempio illustrato dalla figura 3.6. Come é possibile vedere allo scadere del timer il Server recupera la nuova chiave di cluster ottenuta facendo l'operazione di Mixing tra l'elemento alla testa della coda (che segue una politica FIFO(First In first Out)) della catena associata al nodo interno 8 e la chiave di join corrente e lo invia alle mote 0 e alle mote1. Si ricordi che questa chiave di cluster permette di garantire la Forward Security e la Backward Security.

Il KM riceve dai vari Pc le richieste per il recupero della chiave di cluster del padre del sensore con cui vuole comunicare.

Ogni Pc segue una politica stabilita dal KM in base alle quale puó essere autorizzato a inviare componenti alla mote.Il PC é caratterizzato da un livello di privilegio che puó essere Normale o di Emergenza. Nel caso in cui ha livello di privilegio normale il KM lo autorizza a caricare componenti solo se la rete si trova in una condizione normale e non di pericolo. Nel caso in cui ha livello di privilegio emergenza puó effettuare caricamenti quando vuole, poi-

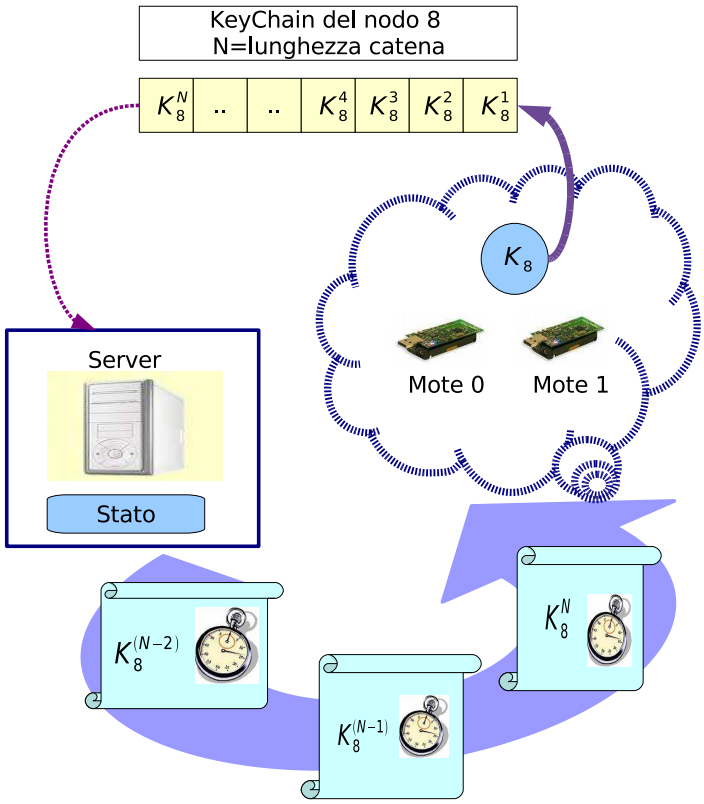


Figura 3.6: Esempio di modifica della chiave della catena associato al nodo interno 8

ché la componente caricata servirá per fronteggiare situazioni di pericolo. Il KM per proibire il caricamento della componente invierá al PC la vecchia chiave di cluster. Invece se deve permettere il caricamento invierá la chiave di cluster aggiornata. Questa politica decisionale permette di stabilire una prioritá dei compiti da eseguire per realizzare gli obiettivi di una *WSN* usata per una specifica applicazione. Successivamente il Pc invierá alla mote la vera chiave di sessione cifrata con la chiave precedentemente scelta dal KM. Dato che questa chiave viene conosciuta solo dai figli di quel cluster solo loro potranno decifrarla e questo né garantirá la piena confidenzialitá della comunicazione.

L'altro modulo del Server é l'*IPC*. Questo modulo interagisce con il Pc per lo scambio dei messaggi effettuati per decidere quale deve essere la chiave di sessione con la quale successivamente la mote caricherá.

3.2 Lato Pc

Il Pc interagisce con il Server e la mote. É un entitá che ha come attributi:

- una chiave pubblica e e la relativa chiave privata
- un livello di privilegio che puó essere normale o di emergenza

I compiti che svolge sono :

1. sottoprotocollo per lo scambio di una chiave di sessione con il server.
2. invio di una componente dinamica.

3.2.1 Invio di una componente dinamica

Il Pc deve potere inviare una componente in maniera autenticata e confidenziale. Per potere garantire l'autenticitá della componente si potrebbe usare la tecnica della firma digitale. Questa tecnica non é molto conveniente perché é computazionalmente costosa. Quindi si usa una tecnica alternativa che prevede l'uso del meccanismo di hashing. I passi che vengono eseguiti affinché un PC invii una componente sono i seguenti:

(a) La componente é divisa in piccoli pacchetti di dimensione arbitraria. Su ogni pacchetto si calcola l'hash (usando *AES* come algoritmo di hashing secondo lo schema Davies-Meyer) che viene inserita all'interno del pacchetto precedente (figura 3.7).

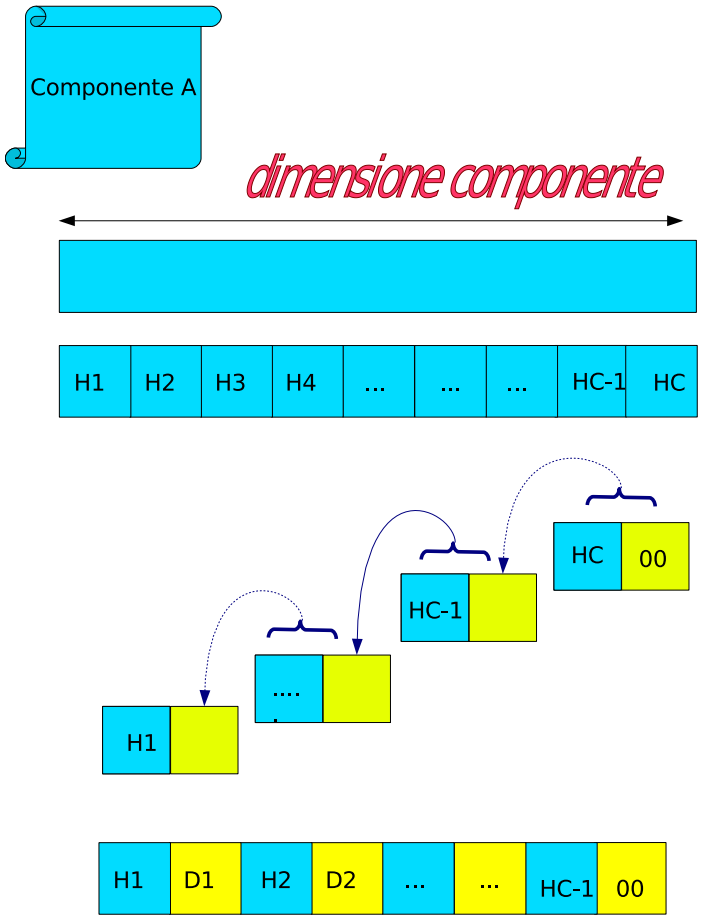


Figura 3.7: Divisione di una componente in pacchetti e calcolo delle hash

La hash dell'ultimo pacchetto é riempita con tutti 0. L'autenticitá del primo pacchetto é ottenuta in maniera diversa (vedere sezione 3.3.1).

(b) Ogni pacchetto *Hi* + *Di* viene cifrato con la chiave di sessione stabilita grazie all'interazione con il Server.Si chiami questa chiave K1.(figura 3.8)

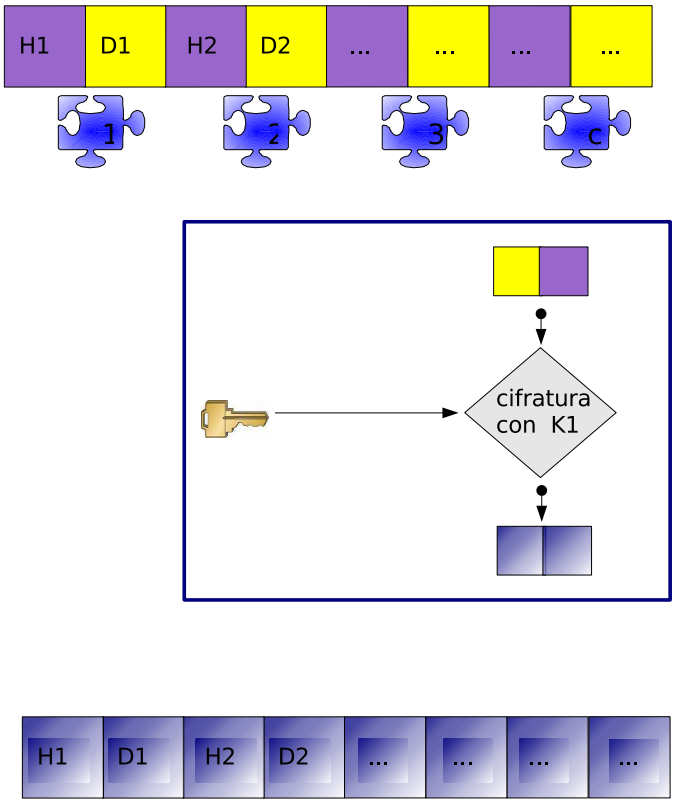


Figura 3.8: cifratura dei pacchetti della componente con la chiave si sessione

(c) A questo punto il Pc invia i pacchetti alla mote (figura 3.9).

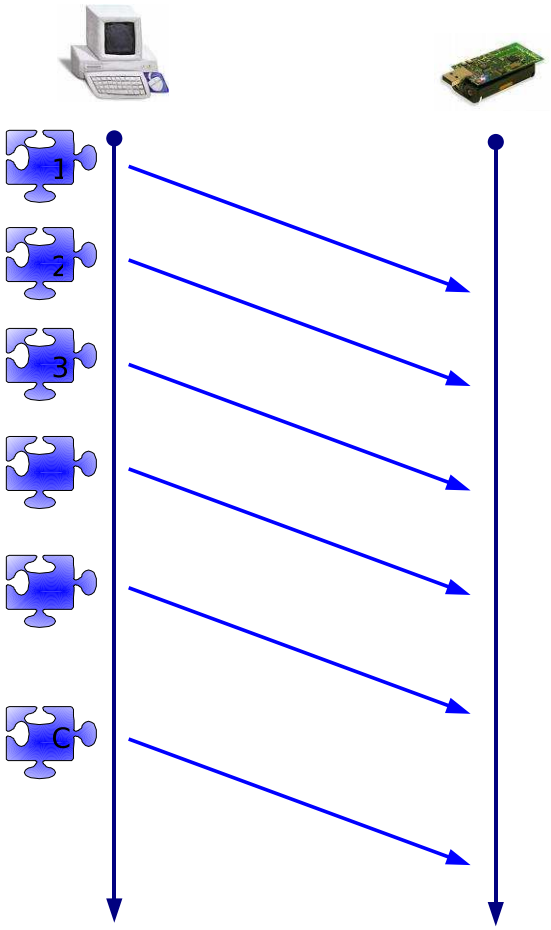


Figura 3.9: Invio dei pacchetti alla mote

3.3 Lato Mote

Ogni nodo sensore é caratterizzato da una chiave privata e una chiave usata per l'autenticazione del primo messaggio. La chiave privata viene generata dal *KDS* facendo un operazione di mixing con la MasterKey e l'identificativo del sensore.Viene usata per interagire con il Server che la usa per cifrare la chiave di cluster del padre. La chiave usata per l'autenticazione del primo pacchetto é condivisa con il PC che l'ha generata costruendo una catena di chiavi seguendo il meccanismo del Key-Chain. La mote é costituita da due componenti che collaborano tra loro secondo un modello di comunicazione ad eventi. Queste componenti sono:

1. ReceiveKey
2. Tcploader

La prima svolge i seguenti compiti:

- Apre una connessione TCP alla porta 6630.
- Si mette in un ciclo di attesa infinita in cui aspetta periodicamente dal server la chiave aggiornata del cluster del padre.

La seconda,invece, svolge i seguenti compiti:

1. Apre una connessione TCP alla porta 6510
2. Recupera la chiave di cluster del padre che é stata precedentemente ricevuta dalla componente ReceiveKey
3. Aspetta che il Pc invii la chiave di sessione cifrata con la chiave di cluster del padre e la decifra
4. Decifra ogni pacchetto ricevuto usando la chiave recuperata e né verifica l'autenticitá vedendo se l'hash del pacchetto ricevuto coincide con l'hash del pacchetto precedentemente memorizzato.
5. Se il processo di autententicazione é andata a buon fine si mette in attesa di autenticare il primo messaggio,altrimenti esce dicendo che non puó autenticare la componente.
6. Invia al Pc un pacchetto di ack che dice se il caricamento della componente é andato a buon fine.

3.3.1 Verifica dell'autenticitá

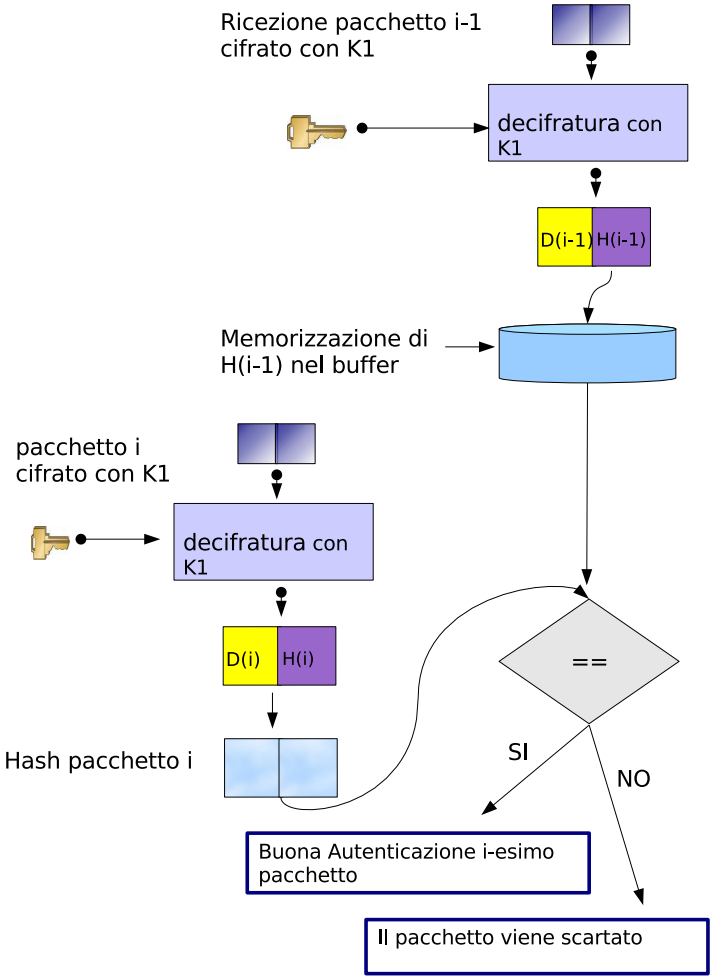


Figura 3.10: esempio di verifica della hash

I pacchetti che il Server invia alla mote hanno la dimensione di 48 bytes. I primi 28 byte sono i dati della componente, mentre gli ultimi 20 sono la hash del pacchetto successivo della stessa componente. Quando la mote riceve un pacchetto, come é possibile vedere nella figura 3.10, né memorizza gli ultimi 20 bytes in un buffer. Quando riceve il pacchetto successivo, nel nostro caso l'i-esimo pacchetto, né fá la hash e lo confronta con il contentuto del buffer. Se sono uguali é andata a buon fine l'autenticazione altrimento no.

3.3.2 Autenticazione primo pacchetto

Per potere autenticare il primo pacchetto il PC e la mote condividono una chiave. Questa chiave é stata creata dal PC eseguendo il meccanismo di Key-chain. Il Pc usa come seme della catena del Key-Chain una chiave incrementale che condivide con tutti gli altri PC e invia alla mote la prima chiave della catena.

Si indichi con $K1_A$ il secondo elemento della catena, con *K0* il primo elemento della catena e con *X0* il primo pacchetto.

Il PC userá il secondo elemento della catena per fare il MAC del primo pacchetto della componente e invierá alla mote $K1A$**+HMAC(X0)**.

La mote,quando riceve questo pacchetto verifica che :

1 Hash($K1A$)=K0

2 $HMAC_{K1_A}(X0) = HMAC\,ricevuto$

Se vengono verificate queste due condizioni puó essere caricata la componente.

3.4 DataBase Chiavi e file StatoPC

Per potere permettere al server di conoscere le chiavi pubbliche dei Pc e ai vari Pc di conoscere la chiave del server é stato creato un Database dove sono state memorizzate le chiavi. Il database si chiama *keydb* e al suo interno é stata creata una tabella chiamata *DBPuBlic*. Questa tabella e' formata da due colonne:

1. ID_{PC} : identificativo del Pc (costituito da 3 cifre).
2. PUBLIKKEY: chiave pubblica del PC in formato Stringa.

La classe *DBBuffer* si occupa della gestione di questa base di dati. Questa classe rappresenta un entitá che deve potere interagire con i Pc per la creazione di buone chiavi pubbliche. Come é possibile vedere dalla prima immagine della figura 3.11, quando il PC effettua una richiesta di generazione della chiave, né viene creata una chiave pubblica e privata appropriata. La chiave pubblica viene inserita all'interno del DataBase e viene inviata indietro al PC. Come é possibile vedere dalla seconda immagine della stessa figura, quando il server effettua una richiesta di recupero della chiave pubblica di uno specifico PC, il DBBuffer cerca la chiave nella tabella DBPublic e, se la trova, la invia indietro al server.

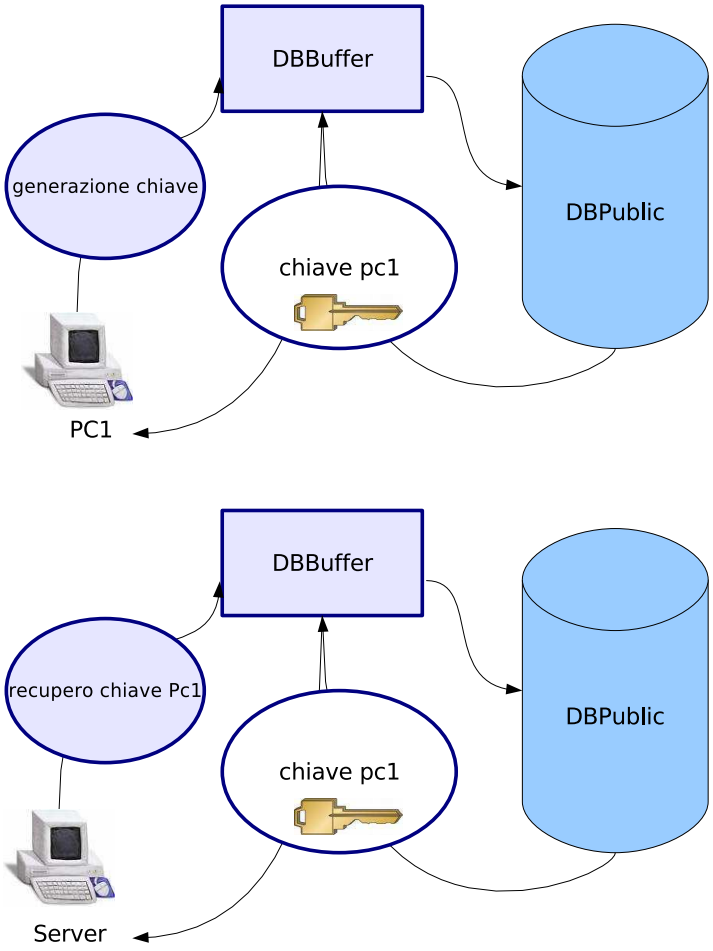


Figura 3.11: operazioni di generazione della chiave e di recupero della chiave

In fase di inizializzazione del Server é stato generato un file chiamato *livpriv*. Questo file permette di stabilire i privilegi che sono stati dati ai vari PC cioé permette di stabilire se il Pc é autorizzato a inviare la componente. É costituito da due colonne:

1. Identificativo PC
2. Stato del PC

Capitolo 4

Protocollo tra il Pc e il Server

4.1 Messaggi del protocollo

Il protocollo tra il Pc e il Server permette stabilire una chiave di sessione usata dal Pc e il sensore per comunicare.Per stabilire questa chiave di sessione si userá sia la crittografia a chiave pubblica che la crittografia a chiave simmetrica.

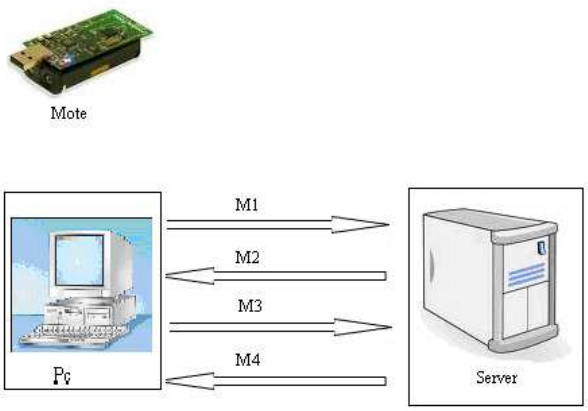


Figura 4.1: Protocollo

La chiave di sessione che verrá scelta sará una chiave che seguirá lo standard *SkipJack* di 16 byte.

Si ipotizzi che vi sia una Certification Authority che assegni una chiave pubblica e privata

(che segue l'algoritmo RSA) al Server e a tutti i Pc che intendono inviare una componenente a una mote della rete.

Il Server condivida con ciascuna mote la chiave di cluster corrente del suo nodo padre.

Sia

K_{PCi}^{-1} la chiave privata del i-esimo Pc

K_{PCi} la chiave pubblica dell'i-esimo Pc

K_{server}^{-1} la chiave privata del Server

K_{server} la chiave pubblica del Server

K la chiave condivisa tra il Server e la mote

K_1 la chiave di sessione generata per permettere la comunicazione tra il PC e il sensore

I meassaggi che vengono inviati sono i seguenti:

$$M1 : Pci \rightarrow Server\{PC, \{KA, NA\}_{K_{PCi}^{-1}}\}_{K_{server}}$$

$$M2 : Server \rightarrow Pci\{\{KB, NA\}_{K_{server}^{-1}}\}_{K_{Pci}}$$

$$M3 : Pci \rightarrow Server\{K1, Nx\}_{K_{con=h(KA,KB)}}$$

$$M4 : Server \rightarrow Pci\{N_x, \{K_1\}_K\}_{K_{con=h(KA,KB)}}$$

4.2 Analisi del protocollo

Si analizzi il protocollo con la logica Ban. É una logica di *beleif* e *action* che serve a provare se un protocollo rispetta i principi tradizionali della crittografia.

L'analisi del protocollo si baserá sui seguenti passi:

1. Determinare le ipotesi di partenza
2. Applicare i postulati a ciascun passo in modo da potere determinare i belief raggiunti dai partecipanti ai vari passi del protocollo.

I postulati a cui si fará riferimento sono i seguenti:

1. Message meaning rule

   (a) $$\boxed{\frac{P| \equiv \ Q \overset{K}{\Longleftrightarrow} P, \ P \triangleleft \{X\}_K}{P| \equiv Q| \sim X}}$$

   Sia K una chiave condivisa tra P e Q e P veda un messaggio cifrato con K contenente X. Nel caso in cui non sia stato P a trasmettere il messaggio, allora crede sia stato Q.

   (b) $$\boxed{\frac{P| \equiv \overset{K}{\longmapsto} Q, \ P \triangleleft \{X\}_{K^{-1}}}{P| \equiv Q| \sim X}}$$

   Se K é la chiave pubblica di Q e P vede un messaggio firmato con K^{-1} contenente X, allora P crede che X sia stato trasmesso da Q.

2. Nonce Verification Rule

   (a) $$\boxed{\frac{P| \equiv \sharp(X), \ P| \equiv Q| \sim X}{P| \equiv Q| \equiv X}}$$

   Se P crede che Q abbia inviato X e P crede che X sia una quantitá fresca allora P crede che Q creda in X. Quindi P é certo che sia stato veramente Q ad inviare X in quella esecuzione del protocollo.

3. Jurisdiction Rule

   (a) $$\boxed{\frac{P| \equiv Q| \equiv X, \ P| \equiv Q \Rightarrow X}{P| \equiv X}}$$

   Se P é sicuro che Q abbia inviato X e che Q sia competente nella gestione di X allora P crede nel messaggio X.

Ipotesi di partenza

1. $Pc_i \mid \equiv \overset{K_{server}}{\longmapsto} Server$

   L'i-esimo Pc crede che K_{server} sia la chiave pubblica del server.

2. Server $\mid \equiv \overset{K_{Pc_i}}{\longmapsto} Pci$

   Il server crede che K_{Pci} sia la chiave pubblica dell'i-esimo Pc.

3. Server $\mid \equiv \sharp(Na)$

   Il Server crede che Na sia una quantitá fresca.

4. Server $\mid \equiv PC_i \Rightarrow Na$

   Il Server crede che il Pc sia competente nella generazione del nonce Na. E' sicuro che il Pc sia in grado di generare una quantitá pseudo-random.

5. Server $\mid \equiv PC_i \Rightarrow KA$

   Il Server crede che l'i-esimo PC sia competente nella generazione di KA.

6. $Pc_i \mid \equiv Server \Rightarrow Kb$

   L'i-esimo Pc crede che il Server sia competente nella creazione di Kb

7. Server $\mid \equiv Pc_i \Rightarrow K1$

   Il Server crede che l'i-esimo Pc sia competente nella generazione di una buona chiave K1.

8. Server $\mid \equiv Pci \Rightarrow Nx$

   Il Server crede che l'i-esimo Pc sia competente nella generazione di una buon nonce Nx.

9. Server $\mid \equiv \sharp(Nx)$

   Il Server crede che Nx sia una quantitá fresca

Analisi

M1: Il server riceve il primo messaggio. Dato che é cifrato con la propria chiave pubblica lo decifra e ottiene il seguente messaggio:

X= $\{KA, Na\}_{K^{-1}_{Pc_i}}$

Poiché per la seconda ipotesi crede che K_{Pc_i} sia la chiave pubblica del Pc puó applicare il postulato di meaning 1.b. Puó concludere che crede che il Pci abbia inviato il messaggio X ovvero:

$$Server| \equiv Pci| \sim (Na, Ka)$$

A questo punto il Server poiché crede nella freschezza di Na puó applicare il postulato di Nonce Verification Rule. Puó affermare che crede veramente che il Pc_i non abbia inviato un replay ed é sicuro che il messaggio sia stato inviato dal medesimo Pc. Poiché vi é un altro postulato che dice che se un nonce é fresco si puó considerare fresco tutto quello che c'é nel messaggio si puó dire che:

$$Server| \equiv Pci| \equiv KA \text{ (2.4)}$$

Il Server crede che il Pc abbia creato KA.

Raggiunto questo obiettivo il server puó applicare il principio di jurisdiction rule. Il Server in base all'obiettivo 2.4 crede infatti che il Pc abbia veramente firmato il messaggio X ed é certo che il Pc sia sia competente nella creazione di nonce freschi.

Quindi puó arrivare a concludere che crede in KA.

Quindi:

$$Server| \equiv KA$$

M2: l'i-esimo Pc riceve il secondo messaggio. Dato che é cifrato con la propria chiave pubblica capisce che é diretto a lui e lo decifra ottenendo il seguente messaggio:

X= $\{KB, NB\}_{K^{-1}_{server}}$

Poiché per la prima ipotesi crede che K_{server} sia la chiave pubblica del server puó applicare il postulato di meaning 1.b. Puó concludere che crede che il Server abbia inviato il messaggio X ovvero:

$$Pc_i| \equiv Server| \sim (Na, Kb)$$

A questo punto il Pc poiché crede nella freschezza di Na,avendo lui stesso generato Na e avendone ricevuto uno uguale a quello che lui aveva creato, puó applicare il postulato di Nonce

Verification Rule.Quindi puó affermare che crede veramente che il Server non abbia inviato un replay ed é sicuro che il messaggio sia stato inviato dallo stesso server. Si puó dire che:

$Pc_i| \equiv Server| \equiv Kb$ (2.5)

Il Pc crede che il Server abbia creato Kb. Il nonce Na ha un altra proprietá che é quella di creare un legame tra il primo messaggio e il secondo messaggio.

Raggiunto questo obiettivo il pc puó applicare il principio di jurisdiction rule. Il PC in base all'obiettivo 2.5 crede che il Server abbia veramente firmato il messaggio X ed é certo che il Server sia sia competente nella creazione di nonce freschi.

In base al principio di jurisdiction rule puó arrivare a concludere che crede in Kb.

Quindi:

$Pc| \equiv Kb$

Adesso sia l'i-esimo Pc che il Server potranno costruirsi la chiave K_{con} con la quale adesso cifreranno i loro messaggi. Questa chiave viene ottenuta facendo l'hash tra KA e KB.

Si sono raggiunti i due obiettivi:

$Pc_i| \equiv Pc_i \overset{K_{con}}{\Leftrightarrow} Server$ (2.6)

Il Pc crede che K_{con} sia la chiave condivisa tra lui e il Server.
$Server| \equiv Server \overset{K_{con}}{\Leftrightarrow} Pc_i$ (2.7)

Il Server crede che K_{con} sia la chiave condivisa tra lui e il PC.

M3: Il Server riceve il terzo messaggio. Poiché in base all'obiettivo 2.7 condivide una chiave di sessione con il Pc, é possibile applicare il primo postulato di meaning rule.

Quindi si puó concludere che il Server crede che il Pc abbia inviato il suddetto messaggio:

$Server| \equiv Pc_i| \sim (K^1, N_x)$

Poiché in base alle ipotesi iniziali il server crede che Nx sia una quantitá fresca si puó applicare il postulato di Nonce Verification Rule.

In base al postulato precedente si puó concludere che il Server crede che il Pc creda nella freschezza di Nx ovvero:

$Server| \equiv Pc_i| \sim \sharp(N_x)$

Dato che il Server crede nella freschezza di N_x allora crederá nella freschezza dell'intero messaggio. Di conseguenza si puó affermare che il Server é certo che:

$$Server| \equiv Pc_i| \sim K^1$$

É possibile applicare il postulato di Jurisdiction e quindi si puó concludere che il Server crede in K1 quindi:

$$Server| \equiv Pc_i \overset{K^1}{\Leftrightarrow} Motex$$

M4: L'i-esimo Pc riceve il quarto messaggio.Poiché in base all'obiettivo (2.6) raggiunto condivide una chiave di sessione con il Server e non ha inviato lui il messaggio, per il primo postulato di meaning puó concludere che :

$$Pc_i| \equiv Server| \sim (Nx, \{K^1\}_K)$$

Quindi il Pc crede che il server abbia inviato il suddetto messaggio. In base al postulato di nonce é possibile anche concludere che il PC crede nella freschezza di Nx. Quindi:

$$Pc_i| \equiv Server| \sim \{K^1\}_K)$$

In base al terzo principio di Jurisdiction si puó affermare, cosí, che il Pc crede nel messaggio e di conseguenza puó inviare la chiave K_1 criptata con la chiave corrente del Key-Chain del padre della mote.

Capitolo 5

Analisi delle classi di utilitá

Si analizzino in dettaglio le classi che servono ai vari attori dello scenario rappresentato:

1. Chiave e GenChiave: si occupa delle generazione delle chiavi
2. DbBuffer: Gestisce tutte le operazioni riguardanti il DataBase delle chiavi dei Pc e del Server
3. Crypting: Implementa tutte le operazioni riguardanti la crittografia

5.1 Chiave e GenChiave

La classe Chiave é costituita da tre membri:

- identificativo dell'utente
- Chiave pubblica di tipo PublikKey
- Chiave privata di tipo PrivateKey

La chiave privata viene memorizzata in un file privato a cui puó accedere solo l'utente che ha richiesto la creazione della chiave. La chiave pubblica viene inserita all'interno della tabella DBPublic.

La classe GenChiave permette la creazione di oggetti di tipo Chiave. La sua funzione principale é presente nel Listing 5.1:

Permette la creazione di una coppia chiave privata e pubblica che segue l'algoritmo RSA.

```
public  Chiave registra(String iduser1){}
```

Listing 5.1: funzione registra

Il package *java.Security* offre alcune interfacce e classi che permettono la generazione di una coppia di chiavi.

L'interfaccia Key é l'interfaccia di tutte le chiavi crittografiche e definisce le funzionalitá comuni a tutte le chiavi. Tutte le chiavi hanno le seguenti caratteristiche:

1. *un algoritmo*: Algoritmo usato per eseguire le operazioni di cifratura e decifratura.

2. *un encoded form*: Codifica esterna della chiave usata quando é necessario trasmettere la chiave fuori dalla *JVM* (quando per esempio bisogna trasmettere la chiave a una terza parte). Possibili codifiche sono *X.509* che serve per rappresentare una chiave pubblica e *PKCS#8* che server per rappresentare la chiave privata.

3. *formato*: Nome del formato dell'encoded form.

PrivateKey e *PublicKey* estendono l'interfaccia *Key*. Esse non contengono dei metodi ma definiscono solo delle funzioni che verranno implementate da diversi provider. Si fará riferimento al provider *BouncyCastle* e alla classe *RSAPublicKey*. Quest'ultima classe permette di implementare le caratteristiche specifiche delle chiavi RSA.

La classe *KeyPair* é usata per trattenere le coppie chiavi pubbliche e private. Fornisce i metodi *getPrivate()* e *getPublic()* per dare la chiave privata e la chiave pubblica.

La classe *KeyPairGenerator* é usata per generare coppie di chiavi. Un oggetto *KeyPairGenarator* puó essere creato usando il metodo statico *getInstance()*. Questo metodo prende come parametro di ingresso l'algoritmo da usare. Prima di potere creare una coppia di chiavi bisogna inizializzare l'oggetto *KeyPairGenerator*. Bisogna generare un oggetto di tipo *SecureRandom* che permette di creare una coppia di chiavi mai prima usata. Bisogna passargli la dimensione della chiave che nel nostro caso sará di 1024 byte. Adesso é possibile chiamare la funzione *generateKeyPair()* che permette di generare la coppia chiave pubblica e privata.

Nel Listing 5.2 é mostrato un frammento di codice per la generazione della chiave.

```
private final static String  ALGORITHMS_DIGEST = "MD5";
Security.addProvider(new bouncycastle.jce.provider.BouncyCastleProvider());
MessageDigest md=MessageDigest.getInstance(ALGORITHMS_DIGEST);
seed=user+(new Date()).toString();
md.update(seed.getBytes());
KeyPairGenerator keygen= KeyPairGenerator.getInstance("RSA","BC");
keygen.initialize(1024,new SecureRandom(md.digest()));
KeyPair pair=keygen.generateKeyPair();
Kpriv=pair.getPrivate();
Kpub=pair.getPublic();
```

Listing 5.2: frammento di codice per la generazione della chiave

5.2 DbBuffer

La classe DbBuffer interagisce con un Driver necessario per connettersi con il DataBase delle chiavi. É stato usato il driver relativo ai server MySql *"com.mysql.jdbc.Driver"*. L'implementazione di quest'ultimo é presente nel file DbBuffer.java. La figura 5.1 rappresenta i metodi pubblici e statici che si possono usare.

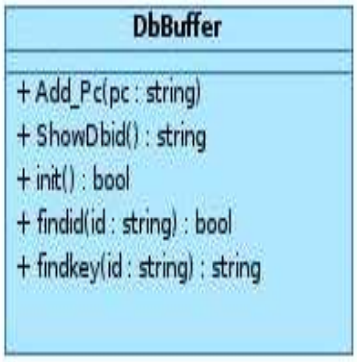


Figura 5.1: rappresentazione uml della classe DbBuffer

La funzione *init()* permette la creazione della tabella *DBPublic* nel caso in cui non é stata ancora creata. Questa funzione per prima cosa cerca il driver di Mysql e né chiede la connessione specificandone la porta e l'utente che né chiede l'accesso. Questo utente deve avere i diritti di accesso richiesti ovvero deve essere presente all'interno di una tabella di Mysql che dice quali utenti possono accedere ai vari Database. Ad ogni utente é possibile associare una password. In questo modo si controllano gli accessi al database, evitandone l'accesso a possibili avversari. Dopo controlla se esiste una tabella di nome *DBPublic*. Nel caso in cui non

esiste crea un'oggetto di tipo *Statement* che permette di eseguire operazioni sul Database ed esegue la query mostrata nel Listing 5.3.

```
SQL_CREATE_DBID="create table DBPublic"+"(ID_PC varchar(32) not null," +
"PUBLIKKEY VARCHAR(324) CHARACTER SET utf8)"
```

Listing 5.3: esempio query per creazione tabella

La funzione *AddPc()* permette l'inserimento di un'altro Pc o di un Server all'interno della tabella *DBPublic*. Questa funzione prende come parametro di ingresso l'identificativo del Pc. Essa controlla se all'interno della tabella é giá presente quell'identificativo. Nel caso in cui é presente esce e manda un messaggio di errore.

Nel caso in cui non é presente crea un oggetto Chiave e un oggetto GenChiave che permette di generare una chiave RSA.(Listing 5.4)

```
Chiave key1=new Chiave();
GenChiave gen=new GenChiave();
key1=gen.registra(ide);
```

Listing 5.4: frammento codice funzione *AddPc()*

La funzione *registra* permette di generare la chiave RSA. Essa appartiene alla classe GenChiave, prende come parametro di ingresso l'identificativo del Pc e ritorna un oggetto di tipo Chiave.

Poi esegue la query mostrata nel Listing 5.5:

```
query= "insert into DBPublic values"+ "(" + "'"+ide +"'"+ ","+
 "'"+pub1 + "');";
```

Listing 5.5: query per l'inserimento di una riga

Questa query permette di inserire all'interno della tabella *DBPublic* una nuova coppia *id* + *chiave pubblica*.

La funzione *ShowDbid()* permette di vedere tutti i Pc e i Server presenti all'interno della tabella. Essa esegue la query mostrata nel Listing 5.6:

```
final String sql_sel="select ID_PC,PUBLIKKEY from DBPublic";
```

Listing 5.6: query per mostrare gli elementi della tabella

La funzione *findid(String id)* cerca una riga della tabella che ha come elemento della prima colonna quello passato come parametro di ingresso. Ritorna true nel caso in cui l'ha trovato.

La funzione *findkey(String id)* cerca una riga che ha come elemento della prima colonna quello passato come parametro di ingresso. Permette di trovare la chiave pubblica associata a uno specifico Pc. Questa funzione ritorna una stringa che rappresenta la chiave pubblica in formato String.

5.3 Crypting

La classe Crypting contiene tutte le funzioni che permettono:

- di trasformare la chiave da codifica byte alla codifica specifica per la chiave pubblica e per la chiave privata
- la cifratura e la decifratura RSA
- la firma e la verifica della firma
- la generazione di un hash
- la cifratura e la decifratura con AES e SkipJack
- la generazione di una chiave simmetrica

Viene implementata nel file Crypting.java. Richiede l'introduzione delle librerie crittografiche di Java.

L'architettura crittografica di Java si articola in due parti:

1. *JCA*: Java Criptography Architecture
2. *JCE*: Java Cryptography Extension

La *JCA* fornisce la cornice generale per la crittografia definendo le classi astratte ma fornendo solo alcune semplici implementazioni di funzionalitá specifiche. Per esempio fornisce le engine class *MessageDigest*, *Signature* e *KeyPairGenerator* che definiscono rispettivamente le funzionalitá di digest, signature e generatore di coppie delle chiavi.

La *JCE* fornisce l'implementazione delle funzionalitá di cifratura e decifratura.

La funzione *PublicKeycod(byte[] inputkey)* prende come parametro di ingresso una chiave codificata in byte e la trasforma in *PublicKey*. *AddProvider()* permette di creare dinamicamente

il provider *BouncyCastle*. La classe *X509EncodedKeySpec* rappresenta la codifica della chiave pubblica. La classe KeyFactory é usata per convertire la chiave nel formato specifico. Un oggetto *KeyFactory* é creato per mezzo del metodo statico *getInstance()*.

```
Security.addProvider(new bouncycastle.jce.provider.BouncyCastleProvider());
X509EncodedKeySpec ks = new X509EncodedKeySpec(inputkey);
KeyFactory kf = KeyFactory.getInstance("RSA","BC");
PublicKey Kpub2 = kf.generatePublic(ks);
```

Listing 5.7: frammento di codice della funzione *PublicKeycod(byte[] inputkey)*

La funzione *decod(byte[] inputkeyprivate)* prende come parametro d'ingresso una chiave codificata in byte e la trasforma in *PublicKey*. La classe *PKCS8EncodedKeySpec* rappresenta la codifica della chiave privata .

```
PKCS8EncodedKeySpec ks1=new PKCS8EncodedKeySpec(inputkeyprivate);
KeyFactory kf = KeyFactory.getInstance("RSA","BC");
PublicKey Kpub2 = kf.generatePrivate(ks1);;
```

Listing 5.8: frammento di codice della funzione *decod(byte[] inputkeyprivate)*

La funzione *RSAEncrypt(String dacrittare,PublicKey keyp)* serve per criptare un messaggio usando come algoritmo di cifratura RSA. L'algoritmo RSA userá una chiave di cifratura di 1024 bit, la modalitá ECB e il padding PKCS1.

Per potere cifrare un messaggio le classi fondamentali sono *Cipher* e *CipherInputStream*. La prima offre funzionalitá di cifratura e decifrazione dei dati mediante uno specifico algortmo.La seconda incapsula il concetto di canale sicuro. Combina un oggetto *Cipher* e *CipherInputStream* per potere gestire automaticamente l'operazione di cifratura.

Il primo passo da eseguire nella funzione mostrata nel Listing 5.9 consiste nel creare l'oggetto *Cipher* configurandolo tramite il metodo *getInstance()*, al quale si passano come parametri di ingresso l'algoritmo usato per la cifratura asimmetrica, la modalitá di suddivisione dei blocchi e il padding di riempimento. A questo punto l'oggetto *Cipher* é inizializzato in modalitá encrypt con la chiave pubblica tramite il metodo *init()*.La modalitá encrypt specifica che sará eseguita l'operazione di cifratura. I dati sono inseriti e cifrati all'interno di un canale sicuro tramite la creazione di un oggetto *CipherInputStream*. Dopo avere effettuato l'operazione di cifratura i dati vengono inseriti in un array di tipo *ByteArrayOutputStream* e poi verranno copiati all'interno di un altro array contenente tutti i blocchi cifrati. Si ricordi che ogni blocco

```
Cipher cipher=null;
byte[] messaggiocrittare=toByte(dacrittare);
byte[] ciphertext=null;
ByteArrayInputStream bIn=null;
CipherInputStream cIn=null;
ByteArrayOutputStream bOut=null;
String ciphermsg=null;
try {
cipher=Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
cipher.init(Cipher.ENCRYPT_MODE, keyp);
bIn = new ByteArrayInputStream(messaggiocrittare);
cIn = new CipherInputStream(bIn, cipher);
bOut = new ByteArrayOutputStream();
int ch;
while ((ch=cIn.read())>=0){
bOut.write(ch);
}
}catch(NoSuchProviderException e1 ){}
catch(NoSuchAlgorithmException e2 ){}
catch(NoSuchPaddingException e3 ){}
catch(InvalidKeyException e4 ){}
catch(UnsupportedEncodingException e5){}
catch(Exception e ){}
ciphertext = bOut.toByteArray();
```

Listing 5.9: frammento di codice della funzione *RSAEncrypt*

cifrato avrá la dimensione di 128 bit. Queste operazioni verranno eseguite fin quando verrá criptato l'intero messaggio.

La configurazione e l'inizializzazione dell'oggetto *Cipher* e la creazione dell'oggetto *CipherInputStream* si trovano all'interno di un blocco try che permette di gestire le eccezioni in caso di possibili errori. In relazione all'errore riscontrato viene propagata l'eccezione corrispondente.

La funzione *RSADecrypt (byte[] Decrypt,PrivateKey priv)* offre la funzionalitá di decifrazione di un blocco. Viene chiamata dalla funzione *DecryptMsg(String msgdecrypt,PrivateKey priv1)* che si occupa di suddividere l'intero messaggio criptato in blocchi da 128 bit.

Come nel caso della funzione precedente, il primo passo da eseguire nella funzione riportata nel Listing 5.10 consiste nel creare un oggetto *Cipher* e configurarlo tramite il metodo *getInstance()*. Successivamente si inizializza l'oggetto *Cipher* in modaltá decrypt con la chiave privata tramite il metodo *init()*. La modalitá decrypt specifica che deve essere eseguita l'operazione di decifrazione. A questo punto il blocco viene inserito e decifrato all'interno del canale sicuro *CipherOutputStream*.

```
Cipher cipher=null;
ByteArrayOutputStream bOut =null;
CipherOutputStream cOut =null;
String plainmsg=null;
try {
cipher=Cipher.getInstance("RSA/ECB/PKCS1Padding","BC");
cipher.init(Cipher.DECRYPT_MODE, priv);
bOut = new ByteArrayOutputStream();
cOut = new CipherOutputStream(bOut, cipher);
cOut.write(Decrypt);
cOut.close();
plainmsg=new String(Hex.encode(bOut.toByteArray()));
}catch(NoSuchProviderException e ){}
catch(NoSuchAlgorithmException e2 ){}
catch(NoSuchPaddingException e3 ){}
catch(InvalidKeyException e4 ){}
catch(UnsupportedEncodingException e5){}
catch(Exception e ){}
```

Listing 5.10: frammento di codice della funzione *RSAEncrypt*

La funzione *firma (byte [] datodafirmare,PrivateKey Kprivpc1)* é usata per fare la firma digitale di un messaggio. Il package *Java.Security* fornisce la classe *Signature* che offre la funzionalitá per eseguire la firma di un messaggio.

```
try{
Signature s = null;
s =Signature.getInstance("SHA1withRSA","BC");
s.initSign(Kprivpc1);
byte[] xxx = null;
s.update(datodafirmare);
xxx = s.sign();
}
catch (NoSuchAlgorithmException e2){}
catch(NoSuchProviderException e ){}
catch (InvalidKeyException e3){}
catch (SignatureException e4){}
```

Listing 5.11: frammento di codice della funzione *firma*

Con la funzione *getInstance()*(riportata nel Listing 5.11) appartenente alla classe *Signature* si crea un oggetto *Signature* specializzato per l'algoritmo *SHAwithRSA* e il Provider *BouncyCastle*. A questo punto si inizializza l'oggetto *Signature* con una chiave private tramite il metodo *initSign()* e gli si forniscono i dati da firmare tramite il metodo *update()*. Con la funzione *sign()* se né fá la firma. Queste istruzioni fanno parte di un blocco try che gestisce le eccezioni.Nel caso in cui la funzione *initsign()* non vá a buon fine scatta l'eccezione *InvalidKeyException* che

dice che la chiave privata per fare la firma non é valida. Nel caso in cui vi sono problemi nella generazione della firma scatta l'eccezione *Signaturexception*.

La funzione *defirma (byte[] msg,byte [] datodadefirmare,PublicKey Kpub1)* verifica la firma di un messaggio.

```
try {
Signature s =Signature.getInstance("SHA1withRSA","BC");
s.initVerify(Kpub1);
boolean res = false;
s.update(msg);
res = s.verify(datodadefirmare);
if (res)
System.out.println("Verifica positiva");
else
System.out.println("Verifica NEGATIVA!!");
} catch (NoSuchAlgorithmException e2){}
catch(NoSuchProviderException e ){}
catch (InvalidKeyException e5){}
catch (SignatureException e6){}
catch (InvalidParameterException  e7){}
```

Listing 5.12: frammento di codice della funzione *defirma*

Come nel caso della funzione precedente anche nella funzione riportata nel Listing 5.12 si crea un oggetto *Signature* e con la funzione *getInstance()* si specifica quale algoritmo verrá usato per fare la verifica della firma. Questa volta con la funzione *initSign()* si reinizializza l'oggetto *Signature* con la chiave pubblica. Se ci sono problemi nella inizializzazione dell'oggetto *Signature* scatta l'eccezione *InvalidKeyException*. Con la funzione *update()* gli si forniscono i dati da verificare. La funzione *verify()* verifica se la firma é andata a buon fine.

La funzione *Hash(String K1, String K2)* calcola l'hashing di due stringhe concatenate. Questa funzione é stata creata per potere generare una chiave che verrá usata nel sottoprotocollo di autenticazione per lo scambio del terzo e quarto messaggio. Il package Java.Security fornisce la classe *MessageDigest* che offre le funzionalitá per fare il digest di un messaggio. La prima operazione da eseguire nella funzione riportata nel Listing 5.13 é quella di creare un oggetto *MessageDigest* tramite il metodo *getInstance()* al quale si dice il provider e quale algoritmo usare per fare l'hashing. L'algoritmo che viene usato per fare l'hashing é *SHA* e il provider é *BounyCastle*. Dopo con la funzione *digest()* se né fá l'hashing.

La funzione *type_encrypt(byte[]text,byte[]key_byte,String typekey)* offre la funzionalitá di cifratura di un messaggio per mezzo di una chiave simmetrica. Questa funzione permette di

```
try {
String hash=null;
MessageDigest sha = null;
sha = MessageDigest.getInstance("SHA-1","BC");
byte[] impronta = sha.digest(messaggio);
hash=new String(Hex.encode(impronta));
} catch (NoSuchAlgorithmException e1){}
catch(NoSuchProviderException e2){}
```

Listing 5.13: frammento di codice della funzione *hash*

potere specificare quale algoritmo usare per fare la cifratura. Verrá usata nei capitoli successivi per potere cifrare messaggi usando come cifrari *Skipjack* e *AES*. La tecnica della crittografia a chiave simmetrica permette di potere cifrare un messaggio in un tempo minore rispetto alla tecnica a crittografia asimmetrica (analizzata precedentemente con la funzione *RSAEncrypt*).

```
Cipher cipher=null;
ByteArrayInputStream bIn=null;
CipherInputStream cIn=null;
ByteArrayOutputStream bOut=null;
SecretKeySpec key = new SecretKeySpec(key_byte,typekey);
byte[] cipherText=null;
try{
cipher =Cipher.getInstance(typekey,"BC");
cipher.init(Cipher.ENCRYPT_MODE, key);
bIn = new ByteArrayInputStream(text);
cIn = new CipherInputStream(bIn, cipher);
bOut = new ByteArrayOutputStream();
inte ch;
try{
while((ch=cIn.read())>= 0){
bOut.write(ch);
}
}catch(IOException e){}
cipherText = bOut.toByteArray();
}catch(NoSuchAlgorithmException e1){}
catch(NoSuchProviderException e2){}
catch(NoSuchPaddingException e3){}
catch(InvalidKeyException e4){}
```

Listing 5.14: frammento di codice della funzione *type_encrypt*

La prima operazione da eseguire (nella funzione riportata nel Listing 5.14) é quella di creare un oggetto di tipo *SecretKeySpec*. Il costruttore di quest'ultima classe permette di costruire una chiave segreta. Prende come parametro di ingresso l'array di byte che deve trasformare in chiave e l'algoritmo che deve seguire per generare la chiave segreta.

Dopo avere creato la chiave segreta, si costruisce un oggetto di tipo *Cipher*. Questo oggetto sará utile per potere eseguire le operazioni di cifratura. L'oggetto *Cipher* é configurato tramite il metodo *getInstance()* al quale bisogna passare come parametri di ingresso l'algoritmo da usare per fare la cifratura e il provider. A questo punto bisogna inizializzare l'oggetto *Cipher* creato con la chiave segreta e in modalitá encrypt tramite il metodo *init()*. La modalitá ENCRYPT_MODE indica che deve essere eseguita l'operazione di cifratura. I dati sono inseriti e cifrati in un canale sicuro per mezzo della creazione dell'oggetto *CipherInputStream*. Dopo che i dati vengono criptati sono inseriti all'interno di un array di tipo *ByteArrayOutputStream*.

La funzione *type_decrypt(byte[]ciphtext,byte[]key_byte,String typekey)* permette di effettuare l'operazione di decifrazione di un messaggio per mezzo di una chiave simmetrica.

```
SecretKeySpec key = new SecretKeySpec(key_byte, typekey);
try{
Cipher cipher = Cipher.getInstance(typekey,"BC");
cipher.init(Cipher.DECRYPT_MODE, key);
ByteArrayOutputStream bOut = new ByteArrayOutputStream();
CipherOutputStream cOut = new CipherOutputStream(bOut, cipher);
cOut.write(ciphtext);
cOut.close();
}catch(NoSuchAlgorithmException e1){}
catch(NoSuchProviderException e2){}
catch(NoSuchPaddingException e3){}
catch(InvalidKeyException e4){}
catch(IOException e5){}
```

Listing 5.15: frammento di codice della funzione *type_decrypt*

Per prima cosa nella funzione riportata nel Listing 5.15 si instanzia un oggetto di tipo *SecretKeySpec* al fine di creare la chiave segreta. Poi bisogna creare un oggetto *Cipher*. Configurarlo tramite il metodo *getInstance()* e inizializzarlo per mezzo del metodo *init()*. Al metodo *init()* si passa la chiave segreta da usare e la costante DECRYPT_MODE che indica che deve essere eseguita un operazione di decifrazione. I dati vengono inseriti e decifrati in un canale sicuro di tipo *CipherOutputStream*.

La funzione *generateKeysessionepluscodifica(String Hash,String messaggio)* implementa la tecnica di cifratura di un messaggio basata su password.

Questa tecnica di cifratura usa algoritmi meno potenti di Blowfish o TripleDes. Questi algoritmi hanno le seguenti caratteristiche:

- usano chiavi fino a 448 bit

- la password di un utente medio é di circa 6 caratteri e quindi di 48 bit
- come password si tende ad usare parole con un determinato significato

Le password sono soggette agli attacchi con dizionario, in difesa dei quali si possono adottare due tecniche:

- Salting: consiste nell'aggiungere alla password un insieme di bit casuali per ampliarne il keyspace.
- Conteggi di ripetizione: consiste nell'effettuare molte volte un'operazione sulla password per ottenere la chiave per il cifrario PBE.

Questa funzione usa *SHA* per fare l'impronta di un messaggio e *Twofish* in modalitá CBC per codificare il testo.

```
String msgcod=null;
SecretKeyFactory kf=null;
char[] password = Hash.toCharArray();
PBEKeySpec ks = new PBEKeySpec(password);
int ripetizioni = 1000;
byte[] salt = new byte[8];
for(int i=0;i<salt.length;i++) salt[i]=00;
try{
kf=SecretKeyFactory.getInstance("PBEWithSHAAndTwofish-CBC","BC");
SecretKey key = kf.generateSecret(ks);
PBEParameterSpec ps = new PBEParameterSpec(salt, ripetizioni);
Cipher c = Cipher.getInstance("PBEWithSHAAndTwofish-CBC","BC");
c.init(Cipher.ENCRYPT_MODE, key,ps);
byte[] codificato = c.doFinal(messaggio.getBytes("UTF8"));
}catch(NoSuchAlgorithmException e1){}
catch(InvalidKeySpecException e2){}
catch(NoSuchPaddingException e3){}
catch(InvalidKeyException e4){}
catch(NoSuchProviderException e5){}
catch(BadPaddingException e6){}
catch(IllegalBlockSizeException e7){}
catch(UnsupportedEncodingException e8){}
catch(InvalidAlgorithmParameterException e9){}
```

Listing 5.16: frammento di codice della funzione *generateKeysessionepluscodifica*

La prima operazione da eseguire nella funzione riportata nel Listing 5.16 é quella di creare un oggetto di tipo *PBEKeySpec* che permette di convertire un array di char in una chiave PBE. Si crea un oggetto di tipo *PBEParameterSpec* che permette di impostare i parametri della chiave PBE che sono il salt e il numero di ripetizioni.

A questo punto si crea un oggetto di tipo *SecretKeyFactory* indispensabile per la creazione di una chiave segreta. Tramite la funzione *getInstance()* appartenente alla classe *SecretKeyFactory* si specifica quale algoritmo dovrá essere usato per potere generare una buona chiave segreta. In questo caso si userá l'algoritmo *PBEWithShaandTwofish-CBC* e il provider sará *BouncyCastle*. Adesso é possibile generare la chiave segreta di tipo *SecretKey* usando la funzione *generateSecret()*. A questa funzione che fá parte della classe *SecretKeyFactory* bisogna passare come parametro di ingresso la chiave PBE precedentemente creata.

Si puó instanziare un oggetto *Cipher* con la funzione *getInstance()*. Questa funzione ha come parametri di ingresso l'algoritmo che dovrá usare e il provider. Si inizializza il cifrario con la funzione *init()*. Questa funzione ha come parametri di ingresso:

1. una costante ENCRYPT_MODE che indica che verrá eseguita l'operazione di cifratura
2. la chiave segreta di tipo *SecretKey*
3. l'oggetto di tipo *PBEParameterSpec*

La funzione *generateKeysessioneplusdecodifica(String Hash,String messaggio)* implementa la tecnica di decifrazione di un messaggio basata su password.

```
char[] password = Hash.toCharArray();
PBEKeySpec ks = new PBEKeySpec(password);
SecretKeyFactory kf=null;
int ripetizioni = 1000;
byte[] salt = new byte[8];
fore(int i=0;i<salt.length;i++) salt[i]=00;
try{
kf =SecretKeyFactory.getInstance("PBEWithSHAAndTwofish-CBC","BC");
SecretKey key = kf.generateSecret(ks);
PBEParameterSpec ps = new PBEParameterSpec(salt, ripetizioni);
Cipher c = Cipher.getInstance("PBEWithSHAAndTwofish-CBC","BC");
c.init(Cipher.DECRYPT_MODE, key,ps);
byte[] decodificato = c.doFinal(toByte(messaggio));
}catch(NoSuchAlgorithmException e1){}
catch(InvalidKeySpecException e2){}
catch(NoSuchPaddingException e3){}
catch(InvalidKeyException e4){}
catch(NoSuchProviderException e5){}
catch(BadPaddingException e6){}
catch(IllegalBlockSizeException e7){}
catch(InvalidAlgorithmParameterException e9){}
```

Listing 5.17: frammento di codice della funzione *generateKeysessioneplusdecodifica*

I passi da eseguire per potere eseguire l'operazioni di decifratura (riportate nel Listing 5.17) sono molto simili a quelle eseguite per l'operazione di cifratura.

Vengono eseguite le seguenti azioni:

1. Si crea un oggetto di tipo *PBEKeySpec*.
2. Si crea un *SecretKeyFactory* per produrre una chiave segreta.
3. Si genera una quantitá salt impostata a 0
4. Si usa lo stesso numero di ripetizioni presente nell'operazione di cifratura.
5. Si crea un *PBEParameterKeySpec* considerando come parametri di ingresso il salt e il numero di ripetizioni.
6. Si crea un oggetto di tipo *Cipher* avendo come parametri di ingresso il *SecretKey* e il *PBEParameterSpec*.
7. Si decifra il dato con la funzione *Dofinal()* del *Cipher*.

Capitolo 6

Lato Server

Il lato Server é stato implementato in linguaggio Java. É stato scelto Java poiché é un linguaggio altamente portabile e ha la capacitá di potersi adattare a qualsiasi ambiente di esecuzione diverso da quello originale. Java ha un grande numero di librerie e c'é un gran numero di persone che si occupano del suo sviluppo.

Il lato server é stato realizzato per essere multithread ovvero ogni qual volta arriva una richiesta da parte di un Pc crea un thread per poterla soddisfare.

Per permettere la comunicazione tra i *Pc* e il *IPC* o il *KM* e il *KDS* sono stati usati dei socket TCP.Per permettere la cifratura dei messaggi sono stati usati gli algoritmi *RSA*, *AES* e *Skipjack*.

Per l'implementazione del lato server sono stati usati e modificati dei moduli precedentemente implementati dall'Ing.Marco Dell'Unto.

6.1 KeyManager

La classe KeyManager era giá presente nel progetto S^2RP. É stata creata per potere svolgere alcune funzioni tipiche del *KDS* quali la creazione e l'inizializzazione dell'Eviction-Tree. Questa classe é stata modificata ed é chiamata dalla componente KM del Server.

All'avvio del sistema, il KeyManager esegue la fase di inizializzazione in cui deve generare l'albero e le relative catene di chiavi. Successivamente trasmette le chiavi correnti ai soli sensori presenti.Successivamente si mette in attesa di ricevere un array di byte chiamato *evento* che lo informa di quale operazione eseguire.

Questo array (riportato nel Listing 6.1) ha la seguente struttura:

```
evento[0]=48;
evento[1]=(byte)index;
evento[2]=pcid[0];
evento[3]=pcid[1];
eventoa[4]=pcid[2];
```

Listing 6.1: buffer evento

Il primo elemento dell'array *evento* contiene un flag che indica che azioni deve eseguire il *KeyManager*.

Il secondo elemento contiene l'identificativo del sensore.

Gli altri elementi dell'array contengono l'identificativo del PC.

Il *KeyManager* puó eseguire le seguenti azioni:

- operazione di join: operazione per l'inserimento di un nuovo sensore (evento[0]=12).
- operazione di left: operazione per l'allontanamento di un nodo dal gruppo (evento[0]=24).
- operazione di refresh: operazione di reinizializzazione di tutti i sensori dell'eviction-tree (evento[0]=36).
- operazione di invio all'*IPC* della chiave del padre del sensore richiesta (evento[0]=48).
- operazione di cambiamento dello stato del server. Questa operazione viene indicata con la costante (evento[0]=50).

Gli eventi vengono generati dalle classi seguenti:

1. *EventStub1.java*: Si mette in attesa di ricevere dall'*IPC* una richiesta di invio della chiave associata al nodo del sensore con cui uno specifico PC vuole comunicare. Quando né viene fatta una richiesta crea un array di byte e lo invia al KeyManager.
2. *ChangeTimer.java*: Si occupa di cambiare periodicamente lo stato del Server da Normale a Emergenza o viceversa.
3. *EventStub.java*: Permette di eseguire le operazioni di join e di left.
4. *MioTimer.java*: Si occupa di fare eseguire periodicamente un operazione di refresh.

Queste classi sono dei thread che vengono mandati in esecuzione dal KeyManager all'avvio del sistema.

6.2 Analisi delle componenti del Server

Il Server é costituito dalle componenti *IPC* e *KM*.

6.2.1 IPC

La classe *IPC.java* permette l'implementazione della componente *IPC*. Si occupa dell'interazione tr il Server e il Pc per la creazione di una chiave di sessione K1. Questa chiave segue l'algoritmo di cifratura SkipJack ed é rappresentata per mezzo di un buffer di 16 bytes. Il PC userá questa chiave per cifrare i pacchetti della componente che vuole inviare.

I passi che vengono effettuati sono i seguenti:

1. Si mette in attesa di ricevere il primo messaggio:

   PC + identificativo del sensore con cui il PC vuole comunicare + (KA+NA) + (KA+NA) cifrato con la chiave privata del PC.

   Tutto il messaggio é stato cifrato con la chiave pubblica del Server.

2. Recupera la propria chiave privata RSA che é stata memorizzata nel file ser1.txt, richiamando la funzione *leggifile* presente nella classe Utility. Nel Listing 6.2 é riportato un esempio di chiave privata.

3. Decifra il primo messaggio richiamando la funzione *DecryptMessage* della classe Crypting.

4. Recupera la chiave pubblica del PC richiamando la funzione *searchpublickey(String key)*. Questa funzione permette di cercare la chiave pubblica dell'interno del database DbKey e ritorna la chiave pubblica trovata.

5. Né verifica la firma usando la chiave pubblica del PC che ha precedentemente trovato.

   Per effettuare l'operazione di verifica della firma richiama la funzione *defirma* implementata nella classe *Crypting*. Questa funzione fá una verifica del messaggio.

6. Se la verifica della firma é andata a buon fine, invia il secondo messaggio che ha la seguente struttura:

   (Kb+NA) + (Kb+NA) firmato con la sua chiave privata.

   Tutto questo messaggio é stato cifrato con la chiave pubblica del PC.

```
Private-Key      RSA Private CRT Key
modulus:
ae46001c3a5ce6bc3c3ab267727b0bec1883d1d0b6b09d64964fd8a002eaa429501693ba0839
68b56ce7ad0228ba38c3b77efd9d0eb8eee28b412d33de0ac124b1a543ceaef59fe869f57827
e2032edc5a4c1342b436b025c23cbbd80b69b7b27cc78f1bc7825cc7f9c82f588af362383cae
b70b66cfce0e24f26fcee577bf73
public exponent: 10001
private exponent:
4e7406d2ae576a2bed3543611c815621a7fb97b5ccb87f726bd47021c08e5eee963643b4a8c5
27651d2cc28ec944e40eff8934a4c29ef0339e1aee3d7e44fb23493ace157fdc2a6afe1035bf
8c2a26f088db40a5650d668a2725774955f01b2269002e87d9ea995a3d81f2fdcd68f052f40d
511cbecf87561667dfdf99482f09
primeP: ed4c00f6041d973336baf4f98cd7d09d083f00b7fda2b23ea7efc921300259500916
1080620242702e0ecabe9a0d17409199d27f66b4c994b0c37ae7e9cfcce7
primeQ: bc025b7ecc9e351fdaca3b8b8aed32296bdbca7919f0ef82e70a282644d396b1c261
73d8dea794d00d7eadf36c9ea3c97a16a4e42906379c9fd220750689fb95
primeExponentP: a7366872f3817a7b2f34dccd40af0f24a89be4c0c22cb457ac752e908149
16706c86acc81ac826d2efd1a2925ef19c91f647a369fd10a5ff6df3c196cc4e8ebf
primeExponentQ: 9b80dfa88b19b81af6de1a4e1ac4819edd0557fcbe4617f48fa049010e20
6c1def06a7c912b04d57a031cd32effe6ebe306b164b58db39982460aa53e0eab3d9
crtCoefficient: 5c9604c62499eae30bd4a8808222a6da2359508e0f2c0e1b326de08f8282
00e3bd9730443155a9a9b0779bb9fa536502c6959cd8a15e968ca1068f6dbffd71c
```

Listing 6.2: esempio di chiave privata

Kb é una quantitá pseudo-random generata per mezzo di un oggetto di tipo *SecureRandom*. La classe *SecureRandom* é presente nella libreria Security di Java.Nel Listing 6.2 é riportato il codice per la generazione della quantitá pseudo-random *KB*. La funzione *getInstance()* della classe SecureRandom dice al generatore di usare come algoritmo *SHA1PRNG*. La funzione *generateSeed* permette di generare un dato seme da passare al generatore. La funzione *setSeed()* permette di inizializzare il generatore con il seme creato e la funzione *nextBytes* genera la quantitá pseudo-random..

```
random = SecureRandom.getInstance("SHA1PRNG");
byte seed[] = random.generateSeed(30);
random.setSeed(seed);
random.nextBytes(KB);
```

Listing 6.3: frammento di codice per la generazione di *KB*

7. Si costruisce la chiave simmetrica K_{con}=Hash(KA||Kb) con cui il server decifrerá il terzo messaggio ricevuto. Questa chiave viene ottenuta eseguendo la funzione *Hash* facente parte della classe *Crypting*.

8. Aspetta l'arrivo del terzo messaggio che ha la seguente struttura:

   (K1+Nx) cifrato con la chiave simmetrica K_{con}=Hash(KA||Kb).

9. Decifra il terzo messaggio con la chiave K_{con} chiamando la funzione *generateKeysessioneplusdecodifica()* presente nel file Crypting.

10. Invia una richiesta alla classe `eventstub1` con la quale richiede la chiave corrente della catena del sensore.

11. Si mette in attesa di ricevere la chiave richiesta dal KeyManager. Per fare questo ha precedentemente creato un *ServerSocket* che accetta richeste alla porta 7720 e uno stream di ingresso di tipo ObjectOutputInputStream.

12. Cifra *K1* con la chiave ricevuta. L'*IPC* usa come algoritmo di cifratura *SkipkackCBCCTS*. Per potere usare questo algoritmo di cifratura si deve istanziare un oggetto di tipo *SkipjackCBCCTS* che né crea un riferimento. Dopo chiama la funzione *SkipjackCBCCTSCipher()* (presente nella classe *SkipjackCBCCTS*) che permette di potere cifrare il messaggio.

13. Costruisce l'ultimo messaggio che inviérá al Pc.Questo messaggio ha la seguente struttura:

    Nx + K1 cifrato con la chiave ricevuta.

    Tutto il messaggio viene cifrato con K_{con}.

6.2.2 KM

La componente *KM* é implementata attraverso la classe *KM.java*. Questa classe si occupa della creazione del file livpriv che assegna ad i vari PC, (che hanno l'identificativo che vá da 100 a 999) un livello di privilegio che puó essere normale o di emergenza.

Crea un oggetto di tipo KeyManager. Dato che quest'ultima classe é stata programmata come thread il KM né crea un thread e lo manda in esecuzione (come viene riportato nel Listing 6.3).

```
int livelli=4;
int arieta=2;
Buffer buf=new Buffer();
BufferPolKey pol=new BufferPolKey();
prova=new KeyManager(buf,arieta,livelli,pol);
Thread cont=new Thread(prova);
cont.start();
```

Listing 6.4: frammento di codice per la generazione della chiave

Capitolo 7

Lato PC

Il modulo *PC* é stato inserito all'interno del progetto della Comunitá Europeo RUNES *ist-runes.org* É stato creato per potere modificare l'utility *"codeprop"* presente nella vecchia versione del Contiki. Questa utility era implementata in linguaggio C e veniva caricata in maniera statica. Permetteva di inviare una componente RUNES autenticata a una mote attraverso una porta USB. Si é fatto il porting di questa utility come componente RUNES che é possibile caricare a RUN-TIME. La nuova componente oltre a supportare le funzioni precedenti offre la possibilitá di potere cifrare la componente da inviare.

Il lato PC é stato implementato in linguaggio Java. É costituito da quattro componenti. La prima componente si chiama *PC* e interagisce con il *Server* per lo scambio della chiave di comunicazione. Le altre componenti sono *AuthLoader*, *SecNet* e *Skipjack* che interagiscono tra di loro per permettere l'invio di una componente in modo autentico e sicuro.

7.1 Componente PC

La componente *PC* é implementata nella classe *PC.java* come un Thread. Si occupa di interagire con il Server per la creazione di una chiave di comunicazione e di permettere il caricamento di una componente. I passi che esegue sono i seguenti:

1. Nel caso in cui il PC non ha una chiave pubblica e la relativa chiave privata genera una chiave che verrá memorizzata all'interno di un file. Altrimenti continua ad eseguire le operazioni successive.

2. Chiede l'identificativo del sensore e costruisce due quantitá pseudo-random *KA* e *NA*.

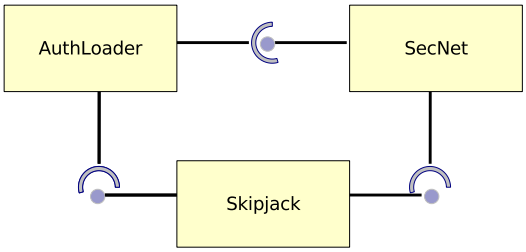


Figura 7.1: Lato Pc:Compenenti usate per l'interazione con la Mote

3. Crea un socket TCP per potere comunicare con il Server.

4. Recupera la propria chiave privata aprendo il file dove é stata memorizzata e la usa per generare la firma di *KA+NA*. Per eseguire l'operazione di firma chiama la funzione *firma* presente nel file *Crypting*.

5. Recupera la chiave pubblica del *Server* accedendo al DataBase *KeyDb* tramite la chiamata della funzione *findkey()* presente nella classe *DBBuffer*.

6. Costruisce il primo messaggio da inviare al server che ha la seguente struttura:

   identificativo Pc + indice del sensore + (KA+NA) + (KA+NA) firmato con la sua chiave privata

   Tutto il messaggio é cifrato con la chiave pubblica del Server.

7. Si mette in attesa di ricevere dal *Server* il secondo messaggio:

   (KB+NA)+(KB+NA) firmato con la chiave privata del Server

   Tutto il messaggio é cifrato con la chiave pubblica del Pc.

8. Dopo avere ricevuto il messaggio lo decifra usando la propria chiave privata richiamando la funzione *DecryptMSg* presente nel file Crypting.

9. Verifica la firma di (KB+NA). Se la firma é andata a buon fine continua ad eseguire le azioni successive altrimenti esce mandando un messaggio di errore a video.

10. Crea la chiave simmetrica K_{con}=hash(KA,KB) che userá per cifrare il terzo messaggio e per decifrare il 4 messaggio che riceverá dal Server.

11. Genera una chiave simmetrica di comunicazione per pemettere l'operazione di cifratura e decifrazione dei pacchetti della componente. Questa chiave userá come algoritmo di cifratura *Skipjack* e sará di 16 byte. Verrá creata richiamando la funzione *GenKey* presente nel file Crypting a cui passerá una quantitá random e l'algoritmo Skipjack.

12. Genera una quantitá random *Nx* e costruisce il terzo messaggio:

    K1,Nx

    Cifra il messaggio con la chiave simmetrica K_{con} che ha precedentemente creato.

13. Invia il messaggio al Server.

14. Si mette in attesa di ricevere dal Server il 4 messaggio:

    Nx, (K1 cifrato con la chiave corrente della catena del padre del sensore)

    Tutto il messaggio viene cifrato con la chiave simmetrica K_{con}

15. Decifra il messaggio ricevuto richiamando la funzione *generateKeysessioneplusdecodifica()*

16. Verifica che la quantitá random *Nx* ricevuta sia uguale a quella che ha generato. Nel caso in cui sono uguali conclude dicendo che il processo di autenticazione é andata a buon fine altrimenti esce mandando a video un messaggio di errore.

17. Crea un oggetto di tipo *CodeproJava* che permetterá di potere caricare una componente a cui passo la chiave K1 cifrata con la chiave corrente della catena del padre al sensore.

7.2 AuthLoader,SecNet e SkipjackCBCCTS

Le componenti *AuthLoader*, *SecNet* e *SkipjackCBCCTS* vengono create e instanziate tramite la classe *CodepropJava.java* (Listing 7.1). La classe *CodepropJava.java* manda a video una shell con la quale chiede all'utente quale componente vuole caricare e se questa componente deve essere autenticata. Queste informazioni sono inviate alla classe *AuthLoader.java*.

7.2.1 AuthLoader

1. Legge la componente RUNES e la inserisce all'interno di un array di char chiamato *bufferstore*. La componente *AuthLoader* é implementata tramite la classe *AuthLoader.java*. Questa componente é collegata alle componenti *SkipjackCBCCTS* e *SecNet* per mezzo di due recettacoli che le permettono di usufruire dei loro servizi.

   La componente *AuthLoader* fá le seguenti azioni:

2. Controlla se la componente deve essere autenticata guardando il terzo parametro di ingresso della funzione *readcomponent()*.

3. Nel caso in cui la componente deve essere autenticata, inserisce all'interno dell'array *bufferstore*(riportato in figura 7.2) in posizione 11 il carattere S altrimenti il carattere N.

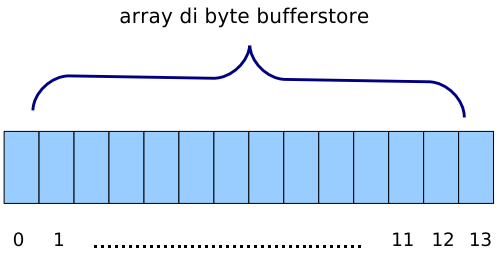


Figura 7.2: array bufferstore

```
Loader loader = new DefaultLoader();
ConnectorFactory cf = new DefaultConnectorFactory(loader);
Capsule capsule = new DefaultCapsule(loader);
ComponentType SecType=capsule.load(new StringPattern("runes.
sampleApp.SecNet"));
Component SecIn = capsule.instantiate(SecType, this);
Interface SecInterface=(Interface) capsule.getAttr(SecIn,"INTERFACE-runes.
sampleApp.ISecNet");
ComponentType AuthLoaderType=capsule.load(new StringPattern("runes.
sampleApp.AuthLoader"));
Component AuthLoaderComponent =capsule.instantiate(AuthLoaderType,this);
ComponentType SkipjackType =capsule.load(new StringPattern("runes.kernel.
cipher.SkipjackCBCCTS"));
Component SkipComponent=capsule.instantiate(SkipjackType, this);
Interface SkipInterface=(Interface)capsule.getAttr(SkipComponent,
"INTERFACE-runes.kernel.cipher.ISkipjackCBCCTS);
Interface SkipInterface1=(Interface)capsule.getAttr(SkipComponent,
"INTERFACE-runes.kernel.cipher.ISkipjackCBCCTS");
Receptacle SecRecept =(Receptacle)capsule.getAttr(AuthLoaderComponent,
"RECEPTACLE-runes.sampleApp.ISecNet");
Receptacle SkipRecept=(Receptacle)capsule.getAttr(AuthLoaderComponent,
"RECEPTACLE-runes.kernel.cipher.ISkipjackCBCCTS");
Receptacle SkipReceptSec=(Receptacle)capsule.getAttr(SecIn,
"RECEPTACLE-runes.kernel.cipher.ISkipjackCBCCTS");
Connector NetReceptSec = capsule.connect(SecInterface,SecRecept, cf);
Connector SeCReceptSki=capsule.connect(SkipInterface,SkipRecept, cf);
Connector SeCReceptSkiSec=capsule.connect(SkipInterface1,SkipReceptSec, cf);
((IAuthLoader)AuthLoaderComponent).readcomponent(name1,ind1,aut,
cript,num1, Key, utente);
```

Listing 7.1: frammento di codice per la creazione delle tre componenti

4. Se la componente deve essere autenticata, la suddivide in blocchi di 28 byte e per ognuno di questi né fa l'hash seguendo l'algoritmo AES. Ogni digest viene memorizzato nel blocco precedente. Di volta in volta costruisce blocchi di 48 byte (nei primi 28 byte é inserito il pacchetto e negli altri 20 l'hash del pacchetto precedente). Ogni blocco di 48 byte viene cifrato usando l'algoritmo SkipjackCBCCTS e la chiave di sessione che é stata stabilita precedentemente grazie al sottoprotocollo di autenticazione tra il Server e il PC. Invia i pacchetti al sensore tramite la funzione *SendSec* presente nella componente *SecNet*, come é riportato nel Listing 7.2.

```
((ISecNet)rSeC.getSingleConnectedInterface()).sendSeC(buildchain1,
indi,filesize,user,K);
```

Listing 7.2: chiamata alla funzione *SendSec*

5. Se la componente non deve essere autenticata, la invia alla mote tramite la funzione *SendNotAutentication* presente nella componente *SecNet*.La funzione *SendNotAutentication* si occupa di suddividere la componente in blocchi di 28 byte. Cifra ogni blocco (usando l'algoritmo SkipjackCbCCTS e la chiave di sessione precedentemente stabilita) che successivamente invierá alla mote (Listing 7.3).

```
((ISecNet)rSeC.getSingleConnectedInterface()).sendNotAuthentication
(bufferstorev,indi,filesize,user,K);
```

Listing 7.3: chiamata alla funzione *SendNotAuthentication*

7.2.2 SecNet

L'implementazione della componente SecNet é presente nella classe *SecNet.java*. La componente *SecNet* é connessa alla componente *AuthLoader* tramite un interfaccia e alla componente *SkipjackCBCCTS* tramite un recettacolo. Quindi fornisce il servizio di invio alla componente *AuthLoader* e usa il servizio di cifratura dei dati offerto dalla componente *SecNet*.

SecNet esegue le seguenti azioni:

1. Crea un socket TCP per connettersi al socket del sensore che ha come indirizzo ip *"172.16.0.1"* e come porta *1012*.

2. Invia la chiave K1 (cifrata con la chiave corrente della catena del padre del senore) al sensore.

3. Se la componente deve essere autenticata:

   - crea la catena delle chiavi (che memorizza nell'array di byte hash_chain) per l'autenticazione del primo pacchetto richiamando la funzione *build_chain()* presente nel file *AES.java*.Ogni chiave della catena ha una lunghezza di 20 byte e la catena é costituita da 20 elementi.
   - Fá partire un timer e allo scadere del timer invia i vari pacchetti della componente.
   - Memorizza il secondo elemento della catena delle chiavi in un buffer chiamato *K1A*.
   - Usa la chiave *K1A* per calcolarsi il *Message Autentication Code* del primo pacchetto della componente richiamando la funzione *hmac()* presente nel file *AES.java*.

Questa funzione ritorna una stringa contenente l'hash del messaggio. Inserisce il valore di ritorno all'interno di un buffer chiamato *hmac_value1*.

- Costruisce il seguente pacchetto:

  hmac_value1+K1A.

- Cifra il pacchetto precedentemente costruito con la chiave K1 e lo invia alla mote che lo userá per potere autenticare il primo pacchetto della componente.

4. Se la componente non deve essere autenticata prende l'array di byte che ha ricevuto dalla componente AuthLoader e lo suddivide in blocchi di 28 bytes. Cifra ogni blocco con Skipjack e lo invia alla mote.

5. La componente *SecNet* dopo l'invio di tutti i pacchetti della componente (sia nel caso in cui la componente deve essere autenticata che nel caso in cui la componente non deve essere autenticata) si mette in attesa di ricevere dal sensore un messaggio di ack. Questo messaggio di ack é di fondamentale importanza poiché il mezzo di comunicazione é wireless. Quindi il Pc non é sicuro che i vari pacchetti siano arrivati a destinazione o che i pacchetti siano corrotti poiché il canale wireless é per sua natura insicuro e inaffidabile.

7.2.3 SkipjackCBBCTS

La componente SkipjackCBCCTS é implementata per mezzo della classe *SkipjackCBCC-TS.java*. Questa componente offre le funzionalitá di cifratura e decifratura avendo a disposizione una chiave simmetrica e usando come algoritmo di cifratura Skipjack in modalitá CBC-CTS.La modalitá che viene usata per il concatenamento dei blocchi é *CBC* (Cipher Blocking Chaining).

SkipjackCBCCTS é un cifrario a blocchi. I blocchi sono sempre di 64 bits e la chiave é di 80 bits. La modalitá CBC prevede la suddivisione del messaggio da criptare in blocchi di lunghezza arbitraria e fissa. Ogni blocco é cifrato con una chiave simmetrica e concatenato con il blocco successivo. In particolare al blocco di testo precedente viene applicata l'operazione di XOR con il blocco corrente di testo in chiaro prima della normale cifratura con la chiave. Il tutto é successivamente cifrato. Per cifrare il primo blocco del messaggio si esegue l'operazione di XOR tra il blocco in chiaro e una quantitá nota chiamata *IV*(Input Vector). Questa quantitá deve essere conosciuta sia dal sender che dal receiver per potere decifrare in maniera corretta il messaggio. La modalitá *CBC* é eseguita insieme alla tecnica *CTS* (*Chipher Text*

Stealing). Il *CTS* altera il processo di elaborazione degli ultimi due blocchi del testo da cifrare. Questa tecnica si occupa di eseguire in maniera diversa il padding dell'ultimo pacchetto. Se M é la lunghezza effettiva dell'ultimo pacchetto e B é la lunghezza di un blocco si riempiono gli ultimi (B-M) bit dell'ultimo blocco con i primi bit del penultimo blocco e si cifra l'ultimo pacchetto.Si tolgono i bit che sono stati usati per riempire l'ultimo blocco dal penultimo blocco, si eseguono nel penultimo blocco tante operazioni di shift a sinistra per quanti sono i bit che sono stati copiati nell'ultimo blocco e si riempiono gli ultimi bit vuoti del penultimo blocco con tutti 0. Adesso é possibile cifrare il penultimo blocco. L'ultimo blocco viene trasmesso prima del penultimo blocco. Questa tecnica comporta un maggiore delay nella propagazione dell'intero messaggio.

Per la creazione di questa componente é stata usata la classe *SkipjackEngine* facente parte della libreria *crypto* implementata dal provider *BouncyCastle*. La velocitá media per l'operazione di cifratura e decifrazione é uguale e la dimensione del testo cifrato é uguale a quella del testo in chiaro.

Capitolo 8

Lato Mote

8.1 Note implementativo

Il lato mote viene implementato in linguaggio C. La vecchia versione del Contiki prevedeva un utility chiamata *tcploader process* che era eseguita al bootstrap sulla mote ed era progettata per lavorare con l'utility *codeprop*. Il *tcploader process* permetteva il caricamento dinamico di una componente. Questo comportava un grande risparmio di memoria, poiché la componente era caricata solo quando serviva. In questo modulo il *tcploader process* é una componente e usa come meccanismo di hashing invece del tradizionale SHA-1 l'*AES* (Advanced Encryption Standard). Per l'implementazione del lato mote sono stati modificati i seguenti file:

- makefile.c: file che contiene tutti i file da compilare per potere eseguire il lato mote
- mykernel.c: file che permette di configurare l'hardware, configurare e far partire l'IP stack e far partire tutti i processi
- receivekey.c e receivekey.h: file che contiene la componente receiveKey
- tcp_loader.c e tcp_loader.h: file che contiene la componente tcp_loader
- itcp_loader.h: file che contiene l'interfaccia della componente tcp_loader
- Shared1.h: file che contiene gli eventi condivisi
- Cipher.h e Cipher.c: file che contiene le funzioni per la cifratura con Skipjack
- ICipher.h: file che contiene l'interfaccia della componente Cipher

- cc240.c e cc240.h: file che contiene la hash con algoritmo di cifratura AES

8.2 Caratteristiche Implementative

In base alle specifiche del progetto si prevede che la mote possa ricevere dal PC la chiave di sessione con cui poi comunicherá e che possa interagire con differenti PC. Quindi si potrebbe inserire all'interno del codice della mote un buffer dove memorizzare le varie chiavi di sessione che condivide con i vari PC.

Bisogna tenere conto dei vincoli derivanti dalle caratteristiche della mote, quali:

- la capacitá limitata della memoria
- la capacitá di far fronte al caricamento dinamico. Bisogna fare in modo di caricare componenti di dimensioni tali da soddisfare i requisiti del sistema software con cui la mote viene inizializzata.

Non é possibile allocare un buffer dove memorizzare tante chiavi di sessione.Questo occuperebbe una dimensione notevole e non si potrebbe soddisfare il requisito di ottimizzazione della memoria.

Si puó decidere, di conseguenza, di allocare un buffer di dimensione pari a 16 bytes. Allora é in grado di memorizzare una sola chiave di sessione.

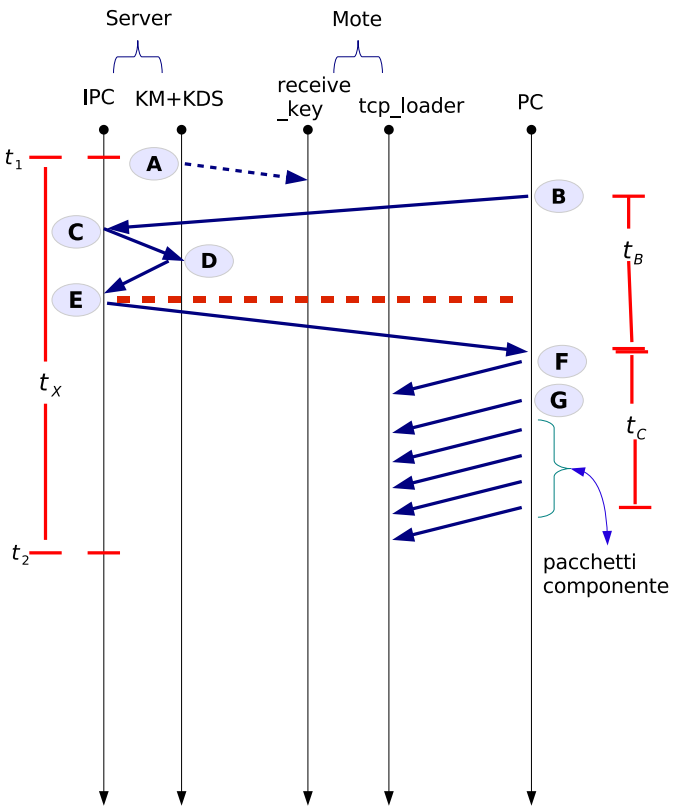
Si é stabilito che la funzione di esecuzione del sottoprotocollo di autenticazione e la funzione di caricamento devono essere sequenziali. Tale scelta é stata fatta poiché la mote puó ricevere componenti da diversi Pc e non si puó prevedere quando interagirá con un PC.

L'esecuzione sequenziale delle due funzioni deve, inoltre, essere eseguita in un tempo tale da potere soddisfare i vincoli derivanti dalle specifiche del progetto:.

1. Il server periodicamente cambia stato. Al variare dello stato cambia la chiave di cluster dei nodi del penultimo livello che bisogna inviare ai corrispettivi nodi sensori figli.
2. Quando viene eseguita un istanza del sottoprotocollo di autenticazione tra uno specifico Pc e il Server, quest'ultimo invia al medesimo Pc la chiave di sessione cifrata con la chiave di cluster corrente del padre della mote.

In base a questi due vincoli bisogna trovare un compromesso tra il tempo stabilito per cambiare lo stato del server e il tempo che passa per eseguire il sottoprotocollo di autenticazione e il tempo per caricare la componente.

Questo é importante per ridurre al minimo la possibilitá che la mote riceva la successiva chiave della catena prima che il Pc non abbia finito di inviare tutti i pacchetti della componente. Questa possibilitá si presenta solo nel caso in cui il Pc non ha ancora inviato nessun pacchetto della componente. Nel caso in cui il Pc ha cominciato ad inviare dei pacchetti della componente la chiave di sessione é stata bene decifrata dalla mote e non puó essere modificata. Infatti é stata memorizzata all'interno di un array che non puó essere modificato fin quando la componente non viene caricata completamente.



Si faccia riferimento allo scenario rappresentato dalla figura precedente. Tale scenario rappresenta l'interazione tra il Server, il Pc e la Mote. Si focalizzi l'attenzione sulla mote.

Sia:

- $t_i \, \forall i \; : 1, ..., N$ l'istante di tempo del cambiamento dello stato del server

- $K_i \, \forall i \, : 1, ..., N$ la chiave i-esima della catena associata al padre della mote
- t_X il tempo stabilito per cambiare lo stato del server
- t_B il tempo impiegato per eseguire il sottoprotocollo di autenticazione
- t_C il tempo per l'invio della chiave di sessione + tempo di caricamento della componente

Le azioni eseguite dal Pc, dal Server e dalla Mote sono le seguenti:

1. A: Il KDS invia la chiave corrente K_i alla componente receivekey della mote
2. B: Inizio del sottoprotocollo di autenticazione tra il PC e l'IPC
3. C: L'IPC richiede al KDS la chiave corrente K_i
4. D: Il KDS invia la chiave di sessione cifrata K_i
5. E: Fine del sottoprotocollo di autenticazione

Affinché il processo di caricamento possa terminare con successo é necessario che:
$\boxed{t_B + t_C \leq t_X}$
É necessario fare in modo che il tempo t_X non sia abbastanza grande.Si verificherebbe la possibilitá che un avversario possa intercettare tanti pacchetti cifrati con la stessa chiave di cluster corrente del padre della mote e quindi una maggiore probabilitá di individuare le diverse chiavi di sessione. Bisogna cercare di ridurre al minimo il tempo che intercorre tra l'invio dei diversi pacchetti della componente per cercare di ridurre il piú possibile il tempo di caricamento di quest'ultima.

8.3 Decriptazione della chiave di sessione

Sulla mote sono presenti due componenti: il receivekey e il tcp_loader che interagiscono tra di loro. Il receivekey contiene un ciclo infinito con cui si mette sempre in attesa di ricevere la chiave di cluster aggiornata associata al nodo padre. Dato che tale processo contiene una receive e la receive é bloccante fin quando viene ricevuto un dato, si é ritenuto indispensabile introdurre un ulteriore componente ovvero il tcp_loader che interagisce con il Pc per la ricezione dei pacchetti della componente. Anche la componente tcp_loader contiene un processo

con un ciclo infinito con cui si mette in attesa di ricevere dal Pc la chiave di sessione cifrata con la chiave della catena e i pacchetti della componente cifrati con la chiave di sessione. L'interazione tra le due componenti avviene tramite uno scambio di eventi.

All'interno del file condiviso Shared.h sono presenti due eventi di tipo *process_data_t* event_read e event_read1. L'evento event_read viene allocato nel costruttore della componente tcp_loader, mentre l'evento event_read1 viene allocato nel costruttore della componente receivekey.

Nel momento in cui il tcp_loader riceve la chiave di sessione cifrata con la chiave del padre invia il segnale *event_read* alla componente *receivekey* attraverso la funzione mostrata nel Listing 8.1.

```
process_post(PROCESS_BROADCAST, event_read ,(process_data_t)data);
```

Listing 8.1: invio del segnale event_read alla componente *receivekey*

La componente *receivekey* si é messa in attesa di ricevere un segnale di tipo *event_read* come mostrato nel Listing 8.2.

```
if(ev == event_read)
PROCESS_WAIT_EVENT_UNTIL(ev==tcpip_event && uip_conn->lport==HTONS(6630));
```

Listing 8.2: ricezione del segnale event_read

Nel momento in cui riceve un segnale di quel tipo si blocca in attesa che si verifichi un evento tcp ovvero che arrivi la chiave corrente del padre cifrata con la propria chiave privata aggiornata alla porta 6630 da parte del *KeyManager*. Quando arriva la chiave corrente la inserisce all'interno del buffer statico *proviamo* e invia un segnale di tipo *event_read1* alla componente tcp_loader, che é in attesa di ricevere un evento di quel tipo.In questo modo il tcp_loader riceve la chiave di cluster del padre cifrata con la propria chiave privata.All'interno della componente *tcp_loader* vi sará il buffer mostrato nel Listing 8.3 che rappresenta la chiave privata che le é stata assegnata dal *KeyManager* attraverso l'operazione di *Mixing* tra la MasterKey e il suo identificatore.. Questa chiave verrá usata per potere decifrare la chiave del

```
static u_int8_t private_key[10] = {0x4b,0xd5,0xcf,0xef,0x33,0x85,0x2e,
0x20,0x07,0x1b};
```

Listing 8.3: buffer contenente la chiave privata del nodo

padre, invocando la funzione *CBCMode_decrypt()* presente nella componente *Cipher* a cui il

tcp_loader puó accedere attraverso il recettacolo *cbcmode_r* che ha precedentemente creato. A questo punto il *tcp_loader* dispone della chiave del padre con cui potrá andare a decifrare la chiave di sessione precedentemente ricevuta che andrá a memorizzare nel buffer interno *Key*.

8.4 Decifratura della componente

Per eseguire l'operazione di decifratura dei pacchetti della componente il *tcp_loader* usufruirá delle funzioni presenti nella componente *Cipher*. La componente *Cipher* si occuperá di fornire le funzionalitá di cifratura e decifratura usando l'algoritmo *Skipjack* e una chiave di sessione simmetrica di 16 bytes. I passi che vengono eseguiti per l'operazione di decifratura sono i seguenti(Listing 8.4):

1. inizializzazione del contesto di cifratura con la chiave di sessione
2. chiamata della funzione di decifrazione

```
INVOKE(cbcmode_r, icbcmode,CBCMode_init(&CMcontext,16,decrypteda));
INVOKE(cbcmode_r, icbcmode,CBCMode_decrypt(&CMcontext,received_tcp,
decrypted,ss,iv));
```

Listing 8.4: frammento di codice per la decifrazione di un pacchetto

La funzione *CBCMode_init()* prende come parametri di ingresso un riferimento al contesto di inizializzazione, la dimensione della chiave e un array contenente la chiave.

La funzione *CBCMode_decrypt()* prende come parametri di ingresso un riferimento al contesto di inizializzazione, l'array contenente il messaggio da decifrare, array dove memorizzare il messaggio, dimensione del dato da decifrare e l'input-vector per potere decifrare il primo blocco.

8.5 Autenticazione della componente

Il *tcp_loader* dopo avere ricevuto la prima componente controlla il flag di autenticazione che corrisponde all'11-esimo elemento del primo pacchetto. Nel caso in cui é uguale a "S" vuol dire che la componente deve essere autenticata, altrimenti si occupa di salvare in flash i pacchetti ricevuti.

Per autenticare le componenti usa le funzioni presenti nel file cc240.h. Questo file contiene le operazioni di cifratura dei dati e l'operazione di hashing che usano come algoritmo *AES* (Advanced Encryption Standard). L'*AES* si basa sull'algoritmo Rijndael e sulla crittografia a chiave simmetrica. É un cifrario a blocchi con una lunghezza del blocco e della chiave variabile. Lo standard prevede una lunghezza del blocco di 128 bit (16 bytes) ed una lunghezza della chiave di 128 bit. Questo algoritmo é disponibile su scala mondiale ed é possibile implementarlo su smart-card.Viene eseguito in modo eccellente nella maggior parte delle piattaforme disponibili e le sue richieste di memoria sono basse per cui é efficiente in hardware e in ambienti con poca memoria.

8.6 Operazione di caricamento della flash

Nel caso in cui il processo di autenticazione é andato a buon fine il *tcp_loader* si occupa di memorizzare i pacchetti ricevuti nella flash. Questi pacchetti sono stati temporaneamente salvati nella *ram* con le seguenti funzione (mostrate nel Listing 8.5) presenti nel file "dev/xmem.c". Per prima cosa la funzione xmem_erase permette di liberare una quantitá di memoria

```
xmem_erase(total_size, EEPROMFS_ADDR_CODEPROP);
xmem_pwrite(decrypted,MESSAGE_SIZE,EEPROMFS_ADDR_CODEPROP
+ imagewritten);
```

Listing 8.5: scrittura ram

ram pari alla dimensione passata come primo parametro e a partire dall'indice presente come secondo parametro. Dopo avere liberato una quantitá di memoria che mi permette di potere immagazzinare tutti i pacchetti della componente viene chiamata la funzione *xmem_pwrite* che permette di andare a scrivere la *ram*. Questa funzione prende come parametri di ingresso l'array contenente il pacchetto da memorizzare, la dimensione del pacchetto e l'indice a partire del quale memorizzare i dati.

Per potere memorizzare i pacchetti all'interno della memoria flash bisogna usare le funzioni presenti nel file della cartella di contiki chiamato *core/elfloader.h*. Le funzioni che vengono chiamate sono le seguenti (Listing 8.6):

La funzione *elfloader_unload()* permette di scaricare la flash.

La funzione *elfloader_load()* permette di caricare la memoria flash a partire dalla posizione

```
elfloader_unload();
int s;
s = elfloader_load(EEPROMFS_ADDR_CODEPROP);
if (s==ELFLOADER_OK)
sprintf(msg," ok");
else
sprintf(msg,"err \%d \%s", s, elfloader_unknown);
printf("Program Loading is \%s Normal Mode",msg);
printf("RESULT \%s",msg);
s = strlen(msg);
```

Listing 8.6: caricamento flash

indicata dal parametro di ingresso. Questa funzione ritorna un intero che indica se l'operazione di caricamento é andata a buon fine. I possibili casi sono i seguenti:

1. s=0 ELFLOADER_OK: indica che il caricamento é andato a buon fine
2. s=1 ELFLOADER_BAD_ELF_HEADER : indica che l'header del pacchetto é stato corrotto.
3. s=2 ELFLOADER_NO_SYMTAB: indica che nessuna voce della tabella dei simboli puó essere trovata nel file ELF.
4. s=3 ELFLOADER_NO_STRTAB: indica che nessuna voce della tabella delle stringhe puó essere trovata nel file ELF.
5. s=4 ELFLOADER_NO_TEXT: indica che la dimensione del segmento testo é 0.
6. s=5 ELFLOADER_SYMBOL_NOT_FOUND: indica che nessun simbolo specifico puó essere trovato.
7. s=6 ELFLOADER_SEGMENT_NOT_FOUND: indica che uno dei segmenti richiesti (.data, .bss o .text) non puó essere trovato
8. s=7 ELFLOADER_NO_STARTPOINT: indica che nessun punto di inizio puó essere trovato nella componente caricata.

Capitolo 9

Conclusioni

9.1 Valutazione delle prestazioni

Dopo l'implementazione del sistema, si é fatta una valutazione del tempo che occorre per l'esecuzione del protocollo nel caso in cui si vuole caricare sia una componente autenticata e cifrata che una componente non autenticata ma cifrata.Si sono fatte un campione di 10 simulazioni, poiché i risultati ottentuti erano molto simili tra loro.

Per il calcolo dei tempi si é usato il comando di linux time. Questo comando dá una rappresentazione del tempo nel segunete formato:

- real:tempo reale trascorso espresso in secondi
- user:totale numero di CPU-secondi che il processo ha eseguito in modaltá utente
- sys: totale numero di CPU-secondi che il processo ha eseguito in modalitá kernel

Il valore di user e sys variano a secondo del processore che esegue il task.

Si é ipotizzato di caricare componenti che hanno una dimensione che varia da 1200 a 1500 Bytes.

I risultati ottenuti sono i seguenti:

Dimensione (B)	Numero pacchetti	real(0m0s)	user(0m0s)	sys (0m0s)
1264	46	1m44,594s	0m2,5725	0m0,060s
1468	53	1m53,555s	0m3,264s	0m072s
1476	53	1m49,135s	0m3,556	0m0,088s

Tabella 9.1: tempi caricamento componenti autenticate e cifrate

Dimensione (B)	Numero pacchetti	real(0m0s)	user(0m0s)	sys (0m0s)
1264	46	1m18,47s	0m2,3625	0m0,150s
1468	53	1m23,797s	0m2,256s	0m056s
1476	53	1m25,15s	0m2,246	0m0,078s

Tabella 9.2: tempi caricamento componenti non autenticate e cifrate

Bibliografia

[1] *Reconfigurable Ubiquitous Networked Embedded Systems* www.ist-runes.org

[2] A.Bicchi G.Dini A.Danesi S.La Porta L.Pallottino I.Savino R.Schiavi
A component-based safe and secure platform for heterogenous wireless multi-robot systems

[3] A.Pacini *Note on authentic code distribution on TMote Sky wireless sensor using Contiki OS and RUNES Middeleware*

[4] G. Dini I. Savino *S^2RP: a Scalable and Secure Rekeying Protocol for Wireless Sensor Networks*

[5] P.Costa G.Coulson C.Mascolo L.Mottola G.P.Picco S.Zachariadis *Reconfigurable Component-based Middleware for Networked Embedded Systems*

[6] *Contiki 2.x Reference Manual* www.moteiv.com

[7] M.Debbabi M.Saleh C.Talhi S.Zhioua *Embedded Java Security* Springer-Verlag London Limited 2007

[8] M.Pistoia F.Reller D.Gupta M.Nagnur K.Ramani *Java2 Network Security* Prentice Hall 1999

[9] *AES* www.dia.unisa.it/ ads/corso-security/www/CORSO-0001/AES/index.htm

[10] *Crittografia in Java* http://nicchia.ingce.unibo.it/oop/web/sicurezza/5-crittografiajava.html

[11] Lamport *Technical note: Password Authentication with insecure communication. CACM: Communications of the ACM* 1981

[12] J.Menezes P.C van Oorschot, A.Vanston *HandBook of Applies Cryptography* CRC PRess 1996