Relazione WORTH

Niccolò Campitelli – 583283

SCELTE IMPLEMENTATIVE

Il **server** è SINGLE-THREADED e utilizza un SELECTOR per la gestione delle connessioni dei clients, registrando i SOCKET per le ricezioni delle richieste in modalità NON BLOCCANTE.
Il **client** utilizza un THREAD per ogni progetto di cui è membro che ne gestisce la CHAT e permette di ricevere messaggi provenienti da più gruppi in contemporanea.
Le **comunicazioni** tra client e server sono codificate con RSA/AES in modo da impedire il furto di dati sensibili e attacchi "Man In The Middle".
Le **serializzazioni** di oggetti sono effettuate senza librerie esterne.
La **lista** di **utenti registrati** viene consegnata al client dopo il login come risposta alla registrazione per le CALLBACK RMI.
L'**interfaccia** del progetto è a linea di comando.

ORGANIZZAZIONE FILE-SYSTEM

SERVER	/home	Directory radice per il server.
	/home/_ip	Directory con file "**next_ip**" contenente il primo indirizzo IP Multicast locale non ancora assegnato ad un progetto e "**free_ip**" contenente la lista di IP liberi dei progetti cancellati.
	/home/_keys	Directory con i file "**pub_key**" e "**priv_key**" contenenti rispettivamente le chiavi RSA pubblica e privata del server.
	/home/users	Directory contenente un file per ogni utente registrato avente lo stesso nome dell'utente. Tali file contengono l'hash della password dell'utente (SHA-256).
	/home/projects	Directory contenente una sotto-directory per ogni progetto creato (e non ancora cancellato) avente il suo stesso nome. Ogni sotto-directory di progetto contiene: - il file "**_chat**" con l'indirizzo IP Multicast della chat del progetto e la chiave AES per la codifica e la decodifica dei messaggi della chat; - il file "**_members**" con la serializzazione di un array di stringhe corrispondenti agli username dei membri; - un file per ogni card del progetto avente stesso nome della card e contenente la serializzazione dell'istanza Card.
CLIENT	/home	Directory radice per il client.
	/home/_certificate	Directory con file "**pub_key**" contenente la chiave RSA pubblica del server.

ORGANIZZAZIONE CLASSI

SERVER	ServerMain	Legge i file “priv_key”, “next_ip” e “free_ip”. Carica gli utenti registrati, i progetti e le cards istanziando le rispettive classi e li inserisce nelle apposite strutture. Crea ServerRegister, ServerCallback e li associa ad un REGISTRY pubblicato per il client. Inizializza SOCKET non bloccante in attesa di connessione del client e lo associa al SELECTOR. Inizia il ciclo di attesa di chiavi pronte nel SELECTOR. Un client appena connesso viene registrato nel SELECTOR con allegata una stringa vuota. Un client che invia un messaggio e che ha associata la stringa vuota deve effettuare il login: in caso di successo gli sarà allegato lo username e potrà utilizzare gli altri servizi. In caso di login, logout o disconnessione il server effettua una CALLBACK a tutti i client registrati al ServerCallback per informarli del nuovo stato dell’utente. Se un utente crea un progetto o ne diventa membro, il server effettua una CALLBACK per inviargli i dati della chat (se online) e istanziare un ClientChat Thread che la gestisca. Se un utente cancella un progetto, i membri online ricevono una CALLBACK per la chiusura dei relativi ClientChat.
	ServerRegister	STUB RMI pubblicato per il client. Presenta un metodo che permette la registrazione di un nuovo utente. Il client comunica username e password. Il metodo restituisce un messaggio testuale con l’esito dell’operazione. In caso di successo effettua una CALLBACK verso tutti gli utenti online comunicando lo stato del nuovo utente (offline) e crea il file dell’utente in “/home/users”.
	ServerRegisterInterface	Interfaccia implementata da ServerRegister. Conosciuta dal CLIENT.
	ServerCallback	STUB RMI pubblicato per il client. Presenta un metodo utilizzato dall’utente per consegnare il proprio STUB RMI e ricevere da questo le CALLBACK. L’utente comunica inoltre i propri dati di login e riceve una CALLBACK per ogni progetto di cui è membro ottenendo i dati di accesso alla chat. Il metodo restituisce un messaggio testuale che in caso di successo è concatenato con la lista di utenti registrati di cui è riportato l’attuale stato di connessione.
	ServerCallbackInterface	Interfaccia implementata da ServerCallback. Conosciuta dal CLIENT.
	User	Istanza che identifica un singolo utente. Memorizza l’hash della password, la lista dei nomi di progetto di cui è membro, la chiave AES dell’attuale comunicazione utente-server e lo STUB RMI remoto fornito dall’utente per ricevere le CALLBACK.

	Project	Istanza che identifica un singolo progetto. Memorizza la lista di username dei membri, l'indirizzo IP Multicast della chat, la chiave AES per comunicare nella chat e quattro tabelle hash per gli stati delle card che associano al nome della card la corrispettiva istanza di tipo Card. Presenta metodi per manipolare le cards e per l'invio di messaggi nella chat da parte del server.
	Card	Istanza che identifica una singola card di un progetto. Memorizza la descrizione e una lista con la cronologia dei nomi degli stati attraversati dalla card nel progetto. Presenta un metodo per l'aggiornamento della cronologia.
CLIENT	ClientMain	Carica chiave pubblica RSA del server da file. Stabilisce connessione TCP col server. Recupera REGISTRY pubblicato dal server ed estrae gli STUB RMI per la registrazione di un nuovo utente e la registrazione alle CALLBACK. Inizia il ciclo di lettura di un comando da tastiera, seleziona il metodo specifico da invocare e, se l'operazione non è locale, invia il comando testuale codificato al server e attende una risposta. In caso di login, se la risposta del server conferma l'operazione, viene istanziato ClientCallback, esportato e passato al server tramite il metodo fornito dall'istanza di ServerCallback recuperata dal REGISTRY. Se l'operazione ha successo, si aggiornano le strutture locali usando la lista di utenti registrati ricevuta nella risposta. I metodi per l'utilizzo della chat di progetto recuperano, dato il nome di progetto, l'istanza di ClientChat che la gestisce e invocano i metodi che mette a disposizione.
	ClientCallback	STUB RMI pubblicato e consegnato al server. Presenta tre metodi invocati dal server per effettuare CALLBACK: 1. comunica all'utente loggato il cambiamento di stato di un utente registrato e aggiorna strutture locali; 2. comunica i dati della chat e il nome del progetto del quale l'utente è diventato membro e avvia un ClientChat Thread per la gestione della chat; 3. comunica il nome del progetto del quale l'utente è membro e che è stato cancellato. Il ClientChat Thread che gestisce la chat del progetto viene terminato.
	ClientCallbackInterface	Interfaccia implementata da ClientCallback. Conosciuta dal SERVER.

	ClientChat	Thread che stabilisce connessione Multicast all'indirizzo IP della chat di progetto ricevuto tramite CALLBACK. Si registra per poter ricevere messaggi. Il metodo run() ripete fino ad interruzione l'attesa di messaggi UDP e l'aggiunta di questi ad una lista. Presenta metodi per la lettura dei nuovi messaggi nella lista e per l'invio di un messaggio in chat.
CLIENT & SERVER	Security	Presenta metodi statici per: - generare l'hash di una stringa con SHA-256; - criptare, decriptare e generare chiavi AES; - criptare, decriptare, generare coppie di chiavi RSA e salvarle o importarle da file.
	Util	Presenta metodi statici per verificare se: - un nome utente/progetto o una password sono validi; - il nome di una card o dello stato di una card è valido; - lo spostamento tra due stati di una card è valido.

SICUREZZA

Tutte le comunicazioni critiche sono codificate. Una comunicazione è critica se i dati che trasferisce non devono essere comprensibili o utilizzabili da terzi. Rientrano in questa categoria sia gli scambi di messaggi TCP client-server, sia le chat di progetto, sia il passaggio di argomenti a servizi remoti RMI.

Le **comunicazioni TCP da un client non loggato verso il server** avvengono con crittografia RSA.
Il server ha associata una coppia di chiavi asimmetriche RSA (2048 bit), la cui chiave pubblica è utilizzata come certificato d'autenticità e tutti i client ne detengono una copia.
Questa comunicazione è finalizzata ad ospitare una richiesta di login contente i dati di accesso dell'utente e una chiave simmetrica AES (128 bit) generata dal client. L'esito della login è comunicato in chiaro poiché non costituisce un'informazione critica in quanto l'identità dell'utente si mantiene anonima ed è comprensibile unicamente al server che dispone della chiave privata RSA.

Le **comunicazioni TCP tra un client loggato e il server** avvengono con crittografia AES in entrambe le direzioni e utilizzano la chiave AES fornita dal client nella richiesta di login avvenuta con successo. L'atomicità della richiesta di login con allegata la chiave AES e con successiva codifica RSA rendono impossibili eventuali attacchi "Man In The Middle", in quanto nessun dato è visibile a terzi e non possono essere effettuate sostituzioni di parti della richiesta durante il trasferimento su di un mezzo pubblico. Altrimenti sarebbe possibile per il malintenzionato sostituire la chiave AES, trasferita dall'utente in un secondo momento, con una chiave AES propria ed avere accesso all'intero traffico.
Le comunicazioni dopo il login procedono con crittografica simmetrica perché più efficiente di RSA.

Le **comunicazioni UDP di messaggi nella chat di progetto** avvengono con crittografia AES utilizzando la chiave AES associata dal server al progetto durante la creazione e condivisa con tutti i membri.

Il **metodo RMI per la registrazione di un nuovo utente** di ServerRegister prende come argomenti l'username e la password codificati con la chiave pubblica RSA del server. L'esito dell'operazione è restituito in chiaro poiché, come per l'esito del login, non costituisce un'informazione critica.

Il **metodo RMI per la registrazione al servizio CALLBACK** offerto da ServerCallback prende come argomenti un riferimento remoto di ClientCallback e i dati di accesso dell'utente codificati con la chiave pubblica RSA del server. Il trasferimento in chiaro dell'oggetto remoto prevede lo scenario in cui un malintenzionato, intercettata la trasmissione del client, recupera il riferimento all'oggetto remoto e ne utilizza illecitamente i metodi per effettuare CALLBACK fuorvianti. Per garantire che solo il server possa utilizzare l'istanza di ClientCallback fornitagli dall'utente loggato, tutti i metodi invocati dell'oggetto remoto ricevono argomenti codificati con la chiave AES fornita dall'utente durante il login, garantendo l'autenticità del server e permettendo al client di individuare le CALLBACK fuorvianti.

STRUTTURE DATI

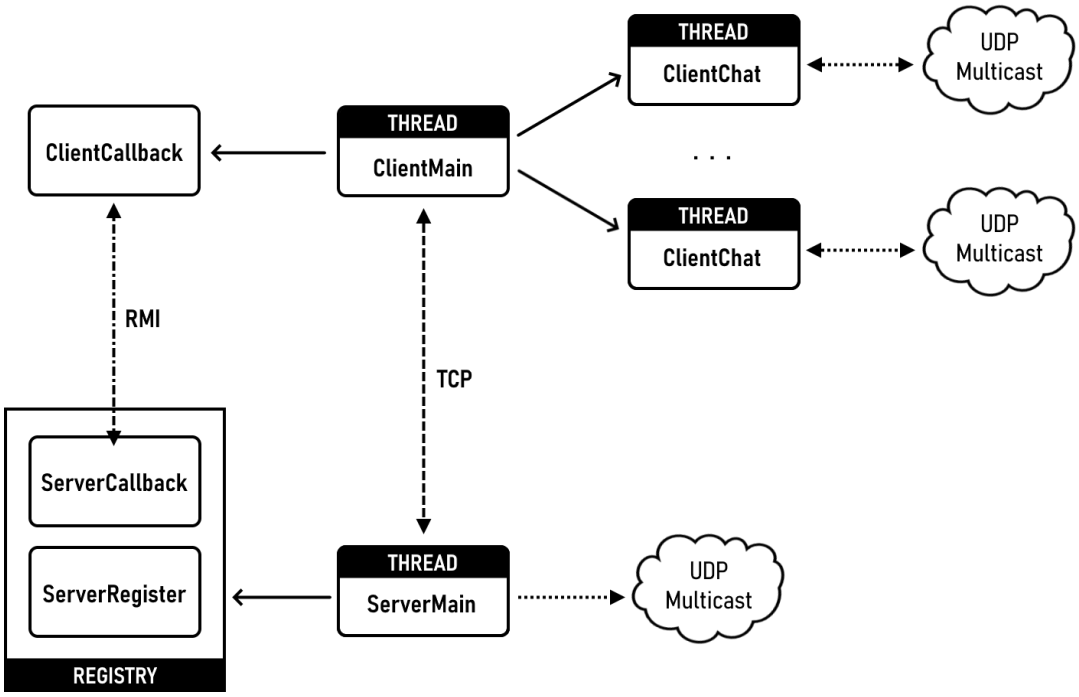
Il **server** crea a Run-Time un'**istanza** dell'omonima classe **per ogni utente**, **progetto** e **card**.

- Gli **utenti** sono memorizzati in **due tabelle hash** che hanno come **chiave** lo **username** e separano gli utenti online da quelli offline.
- I **progetti** sono memorizzati in una **tabella hash** che ha come **chiave** il **nome** del **progetto**.
- Le **cards** sono memorizzate nell'istanza del loro progetto in **quattro tabelle hash** che hanno come **chiave** il **nome** della **card** e le separano in base allo stato di elaborazione.
- Ogni **card** memorizza la cronologia degli stati di elaborazione in una lista interna di stringhe.

Il **client** utilizza:

- **Due liste** con gli **username** degli **utenti registrati** suddivisi tra online e offline.
- **Una tabella hash** che ha come **chiave** il **nome di progetto** di cui l'utente è membro e a cui associa l'istanza di **ClientChat** che ne gestisce la chat.
- **Una lista per ogni ClientChat** che memorizza i **messaggi** ricevuti ma non visualizzati.

THREADS E COMUNICAZIONE



Il **server** si compone di un **singolo thread** e gestisce comunicazioni TCP contemporanee da parte dei clients scansionando un **selector** in cui sono registrati i relativi **socket** in modalità **non bloccante**.
Le connessioni TCP sono quindi raccolte e gestite **una per volta** qualora vengano segnalati più socket con disponibilità in lettura.
Quando il server deve inviare un messaggio UDP ad una **chat** di progetto, crea un **MulticastSocket temporaneo** su cui spedisce il datagramma **senza registrarsi** al gruppo Multicast.
Il server **accetta richieste** di registrazione al servizio **ServerCallback** provenienti dai clients solo se questi sono **attualmente connessi** con TCP e hanno effettuato correttamente il **login**. La **terminazione** della **connessione** TCP di un utente o la ricezione della richiesta di logout **causano** la sua **disiscrizione automatica** dal servizio ServerCallback.

Il **client** si compone di un **singolo thread** per la gestione della comunicazione TCP col server e di **più thread ausiliari** attivati in seguito alla login. Ognuno di tali thread si occupa di **ricevere** e collezionare i **messaggi** di una specifica **chat** di progetto di cui l'utente è membro utilizzando un **MulticastSocket registrato al gruppo** e in ascolto. Non sono creati se l'utente non è membro di alcuna chat e terminano o successivamente al logout o quando un progetto è cancellato.
Quando è richiesto l'invio di un messaggio, viete istanziato un **MulticastSocket temporaneo** per garantire **thread-safety** in caso di ricezioni e invii contemporanei **(1)**; non è registrato al gruppo così da non ricevere messaggi destinati al socket in ascolto.
I membri offline di uno stesso progetto non ricevono i messaggi inviati dagli altri membri o dal server.

CONCORRENZA

SERVER	[HashMap] USERS_ONLINE USERS_OFFLINE	Condivise tra ServerMain, ServerCallback e ServerRegister.
	[HashMap] PROJECTS	Condivisa tra ServerMain e ServerCallback
	[PrivateKey] PRIV_KEY	Condivisa tra ServerMain, ServerCallback e ServerRegister.
CLIENT	[LinkedList] USERS_ONLINE USERS_OFFLINE	Condivise tra ClientMain e ClientCallback.
	[HashMap] CHAT_READERS	Condivisa tra ClientMain e ClientCallback.
	[LinkedList] MESSAGES	Condivisa tra ClientMain e ClientChat.

Le operazioni sulle **strutture dati condivise** tra thread concorrenti sono state incapsulate in blocchi di codice sincronizzati mediante l'uso del costrutto **synchronized(obj){...}** garantendo la mutua esclusione.
L'uso di **metodi synchronized** è stato **evitato** così da circoscrivere unicamente le aree di codice critico e migliorare le prestazioni riducendo il tempo di possessione del lock.

Sono state utilizzate tabelle hash di tipo **HashMap** poiché più **performanti** in quanto non offrono meccanismi interni di sincronizzazione. La **sincronizzazione esterna** risulta più **efficiente** perché le operazioni effettuate sulle HashMap richiedono l'**acquisizione** in mutua esclusione di **più strutture** e i meccanismi interni per la singola mutua esclusione della tabella risulterebbero superflui.

L'analisi del codice ha riscontrato che:

1. L'HashMap USERS_OFFLINE è sempre utilizzata insieme all'HashMap USERS_ONLINE.
2. L'HashMap PROJECTS è sempre utilizzata da ServerCallback insieme ad USERS_ONLINE in modalità only-read ed è modificata unicamente da ServerMain.
3. PRIV_KEY è inizializzata dal server in un contesto single-threaded, prima della pubblicazione dei servizi RMI. Successivamente è acceduta in modalità only-read.
4. Le LinkedList USERS_ONLINE e USERS_OFFLINE sono sempre utilizzate insieme.
5. L'HashMap CHAT_READERS è utilizzata singolarmente.
6. La LinkedList MESSAGES è conosciuta solo da ClientChat ma ClientMain invoca asincronamente dei metodi di ClientChat che operano sul contenuto.

Seguono le seguenti scelte implementative per gli stessi punti:

1. Si utilizza **synchronized(USERS_ONLINE){...}** per incapsulare i segmenti di codice in cui è necessario utilizzare USERS_OFFLINE.
2. ServerCallback e ServerMain utilizzano **synchronized(USERS_ONLINE){...}** per operare su PROJECTS, mentre ServerMain evita l'uso di sincronizzazione quando accede in modalità only-read a PROJECTS. L'HashMap è infatti thread-safe nel caso di letture contemporanee **(2)**.
3. PRIV_KEY non necessita di sincronizzazione in quanto utilizzata in modalità only-read.
4. Si utilizza **synchronized(USERS_ONLINE){...}** per incapsulare i segmenti di codice in cui si opera sia su USERS_ONLINE che su USERS_OFFLINE.
5. Si utilizza **synchronized(CHAT_READERS){...}** per operare sull'HashMap.
6. ClientChat utilizza **synchronized(MESSAGES){...}** per operare sulla LinkedList.

Durante la fase del server di caricamento dei dati da file-system, non sono utilizzati meccanismi di sincronizzazione per le strutture aggiornate in quanto ServerMain è single-threaded e non sono stati resi disponibili né i servizi RMI né il socket per l'accettazione di richieste TCP.
L'uso di **synchronized(USERS_ONLINE){...}** per l'accesso alle HashMap USERS_ONLINE, USERS_OFFLINE e PROJECTS garantisce mutua esclusione anche per le operazioni di modifica delle istanze di User e Project contenute nelle tabelle.

NOMI E PASSWORD

Un **nome utente/progetto/card** è valido solo se contiene tra i 3 e i 24 caratteri e se il primo carattere è una lettera. Deve essere formato solo da lettere minuscole, cifre e dal carattere underscore.

Una **password** è valida solo se contiene tra gli 8 e i 32 caratteri. Deve essere formata solo da lettere, cifre e dai simboli {_,#,@,&,%,$}. Deve contenere almeno una lettera minuscola, una maiuscola, una cifra e un simbolo.

Una **descrizione** di **card** è valida solo se contiene tra i 3 e i 256 caratteri.

GESTIONE FILE-SYSTEM

Il **server** non memorizza informazioni ridondanti su File-System.
Ogni file utente già creato è acceduto in modalità only-read. Lo username è ricavato dal nome del file.
Il nome di progetto è ricavato dal nome della directory in "server/home/projects".
I nomi delle cards sono ricavati dal nome dei files nella directory di progetto che non iniziano per '_'.
Il file "**next_ip**" è aggiornato in seguito alla creazione di nuovo progetto.
Il file "**free_ip**" è aggiornato in seguito alla cancellazione di un progetto o alla creazione di un progetto se contiene almeno un indirizzo IP.

Per ogni directory di progetto:

- Il file "**_chat**" è acceduto in modalità only-read.
- Il file "**_members**" viene sovrascritto solo quando un utente diventa membro del progetto. Contiene la serializzazione di un array String[] con gli username dei membri.
- Il **file di una card** viene sovrascritto solo quando la card cambia stato di elaborazione. Contiene la serializzazione dell'istanza Card composta dalla descrizione testuale e da un ArrayList<String> con la cronologia dei nomi degli stati di elaborazione.

INDIRIZZI MULTICAST

Il range di indirizzi IP Multicast scelto è quello "**Administrative Scoping**": 239.0.0.0 - 239.255.255.255.
Il **TTL** ("Time To Live") dei Datagrammi trasmessi è impostato a "**0**" poiché i clients sono in esecuzione sulla stessa macchina (TCP usa 127.0.0.1) e non è necessario che il traffico raggiunga la rete locale.

Gli indirizzi IP Multicast assegnati a progetti in fase di **cancellazione** sono inseriti in testa alla lista IP_FREE del ServerMain e memorizzati in una nuova riga del file "**free_ip**".

Durante la **creazione** di un nuovo progetto, si verifica se la lista IP_FREE contiene almeno un indirizzo. In tal caso viene rimosso sia dalla lista che dal file e assegnato al progetto, altrimenti si utilizza il nuovo indirizzo Multicast in IP_NEXT, successivamente incrementato (il file "**next_ip**" viene aggiornato). Se è richiesta la creazione di un progetto e gli indirizzi disponibili sono esauriti, viene visualizzato un apposito messaggio di errore e informato il client del disservizio temporaneo.

INPUT E OUTPUT

Il **server** non richiede alcun input da tastiera.
Il **server** visualizza gli esiti della fase di inizializzazione, le operazioni che comportano la modifica del File-System, i cambiamenti di stato degli utenti e le connessioni/disconnessioni dei clients. Vengono visualizzati anche gli eventuali messaggi di errore personalizzati.

Il **client** acquisisce i comandi da tastiera su **singola riga** di testo.
Gli argomenti sono ricavati separando le stringhe dagli spazi. Le **descrizioni** delle **cards** e i **messaggi** della chat possono **contenere spazi**. Il **client** visualizza l'esito della connessione TCP, le risposte ricevute dal server e dai servizi RMI e gli eventuali messaggi di errore personalizzati.
Le password non sono visualizzate durante l'inserimento.

COMPILAZIONE ED ESECUZIONE

Nella directory principale sono presenti 8 files per la compilazione e l'esecuzione automatica del progetto su ambienti Windows e Linux.

I files per **Windows** hanno estensione "**.bat**" e per eseguirli è sufficiente un doppio-click.
I files per **Linux** hanno estensione "**.sh**" e devono essere eseguiti da terminale tramite "./<script.sh>".
Per fornire i permessi necessari agli script utilizzare il comando "chmod 755 <script.sh>".

[win_]compiler	Compila i files ".java" generando i files ".class" del client e del server popolando le rispettive directory.
[win_]reset	Elimina i progetti e gli utenti registrati nel server e tutti i files generati dalla compilazione. Ripristina il file "**next_ip**" col primo IP Multicast locale (239.0.0.0) e rimuove il contenuto del file "**free_ip**".
[win_]run_client	Esegue il ClientMain (è richiesta la compilazione tramite [win_]compiler).
[win_]run_server	Esegue il ServerMain (è richiesta la compilazione tramite [win_]compiler).

Non sono state utilizzate librerie esterne.
Non sono richiesti argomenti per l'esecuzione degli script.
Il comando "**menu**" in ClientMain visualizza l'elenco delle operazioni disponibili con relativa sintassi.
Il comando "**info**" in ClientMain visualizza i dettagli delle connessioni attuali.

RIFERIMENTI ESTERNI

(1) http://www.docjar.com/html/api/java/net/DatagramSocket.java.html

Nel caso di DatagramSocket, l'implementazione dei metodi send() e receive() prevede il costrutto syncronized(p){...} sull'istanza del DatagramPacket passato in ingresso.
MulticastSocket estende DatagramSocket senza override dei metodi send() e receive().
Ne consegue che l'utilizzo di DatagramPacket diversi non è thread-safe in caso di send() e receive() concorrenti sullo stesso MulticastSocket.

(2) https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally.