TP 3: Map Reduce

Taller de Programación

Agustina Barbetta 96528

26 de Abril
1er. Cuatrimestre 2016

1. Introducción

Para realizar esas operaciones de manera eficiente, surge un esquema que las paraleliza: MapReduce. Este esquema consiste en separar un proceso que recibe y devuelve pares (clave, valor) en unas funciones map(), y otras reduce(). En este ejercicio, se implementará un esquema MapReduce simple, en el que los Mappers deberán pertenecer a procesos client, y los Reducers deberán correr en hilos separados de un server. Se deberá deducir, a partir de listas de temperaturas máximas de distintas ciudades del mundo, cuáles fueron las más calurosas del mes de marzo, día por día.

2. Implementación

2.1. Thread

Esta clase abstracta encapsula un `pthread` y contiene la lógica de lanzamiento y unión de hilos. En su módulo también se encuentran las clases `Mutex` y `Lock` vistas en clase.

2.2. Socket

Para llevar a cabo la comunicación entre clientes y servidor, se creó la clase `Socket`.
A las firmas de los métodos que lo requieren se les ha agregado el parámetro `struct addrinfo*`, de esta forma se realiza el llamado a la función `getaddrinfo` por fuera y se pasa sólo un resultado, logrando así un diseño más flexible.
Los métodos de `Socket` suelen encapsular el uso de las funciones de la librería `sys/socket.h`, exceptuando `socket_send()` y `socket_receive()`, las cuales reciben la cantidad de bytes que deben enviar/recibir y permanecen en un ciclo llamando a *send/recv* hasta que la transferencia se haya completado.
Para el manejo de errores se creó la excepción `SocketError`, sabiendo que todas las funciones de la librería `sys/socket.h` devuelven -1 en caso de error, se compara el resultado de cada llamado con la constante `ERROR_CODE` (La cual es -1) y se levanta una excepción. Al crearse, toda instancia de `SocketError` copia la variable global `errno` y guarda su respectivo mensaje en el atributo `msg`.
Para poder guardar los *sockets* en el *stack* y aplicar RAII, se implementó un setter sobrecargando el operador (). De esta forma, el *socket* se crea en el `main` y se pasa por referencia al `WeatherServer`/`WeatherClient` en donde se obtiene el resultado correspondiente de `getaddrinfo` y se termina de configurar.

2.3. AddrInfo

Esta clase encapsula la `struct addrinfo` que se utilizará para los *sockets*, de esta forma se aprovecha el destructor para liberar la estructura. Se utiliza por medio de las clases `ServerAddrInfo` y `ClientAddrInfo` que la tienen por atributo. La diferencia entre estas son los parámetros que pasarán al constructor de `AddrInfo`.

2.4. WeatherClient

Representa al cliente que mapea sus datos del tiempo y envía los resultados al servidor. Su único atributo es una referencia al `Socket` cliente, cuyo *file descriptor* es configurado en el constructor (junto

con el *connect*), luego de obtener la información de `ClientAddrInfo`.
Se sobrecarga el operador () para realizar la lectura del archivo recibido por `stdin`, enviar cada línea al `Map`, recibir el resultado `MapWeather` y enviar este último al servidor.
Finalmente, se realiza el *shutdown* del cliente en el destructor.

2.5. Map

Este *functor* recibe una línea del archivo de `stdin`, identifica día, temperatura y ciudad y devuelve un puntero al par clave valor `MapWeather` resultante.

2.6. MapWeather

Es el resultado del `Map`, sus atributos son un entero día y un `std::make_pair` con un entero temperatura y una `std::string` ciudad.
Su único método `str()` devuelve el par en formato `std::string` estos son de la forma: Día Temp Ciudad\n.

2.7. WeatherData

Consiste en un `map` con enteros día como clave y vectores de `WeatherValue*` como valor. Aquí se guardan todos los pares resultado del `Map` enviados por los clientes `WeatherClient`. Provee métodos para obtener referencias a los vectores de valores correspondientes a cada día así como añadir nuevos `WeatherValue*` a los vectores. Como los hilos `Receiver` lo editan por medio de instancias de `Parser`, se creó una versión protegida del objeto que delega en un `WeatherData` bloqueando y desbloqueando un `Mutex` en su método `push_value()`. Cabe aclarar que `get_days()` y `get_values()` no se protegen, pues no serán utilizados en instancias multi-hilos.

2.8. WeatherValue

Consiste en el valor del par, compuesto por un entero temperatura y una cadena ciudad. estos atributos son provistos en el constructor y no cambian en ningún momento. Existen getters para consultarlos.

2.9. WeatherResults

Consiste en un vector de cadenas, estos serán los resultados de los hilos `Reducer` a imprimir por `stdout`. Provee métodos para su edición y consulta. Como los hilos `Reducer` deben modificarlo, se creó una versión protegida del objeto que delega en un `WeatherResults` bloqueando y desbloqueando un `Mutex` en su método `set_day_result()`. Cabe aclarar que `get_day_result()` no se protege, pues será utilizado luego de unir todos los hilos.

2.10. WeatherServer

Representa al servidor que recibe los pares de cada `WeatherClient`, lanza los hilos `Reducer` y muestra sus resultados por `stdout`. Sus atributos son, una referencia al `Socket` servidor, un `WeatherDataProtected`, un `WeatherResultsProtected` y un vector de `Reducer*`.
Al igual que con el cliente, el *file descriptor* del `Socket` servidor es configurado en el constructor de este objeto (junto con el *bind* y *listen*), luego de obtener la información de `ServerAddrInfo`.
Se sobrecarga el operador () para realizar las siguientes tareas:

- `receive_weather_data()`: Se instancia un objeto `QuitProtected` (un booleano en `false` protegido por un `Mutex`) y un objeto hilo `Acceptor` al cual se le pasa una referencia del `Socket` servidor y una del `QuitProtected`. A continuación se lanza este hilo y se entra en un ciclo que toma caracteres de `stdout` y cambia el estado de `QuitProtected` a `true` cuando detecta una 'q'.
  Al recibir la 'q', se hace un *shutdown* del `Socket` servidor y *join* del hilo `Acceptor`. Al finalizar ese hilo, el `map` encapsulado por `WeatherDataProtected` estará completo con un vector de valores temperatura-ciudad (`WeatherValue*`) por día.

- `launch_reducers()`: Se crea un `Reducer` para cada día, pasándole una referencia del vector de valores correspondiente y se agrega su referencia al vector atributo. Aquí es donde se realiza una simplificación para el SERCOM; se recorre el vector lanzando y uniendo los hilos de a cuatro. Al

finalizar este ciclo, el vector encapsulado por `WeatherResultsProtected` estará completo con la temperatura máxima para cada día.

- `print_results()`: Se recorren los resultados mostrándolos por `stdout`.

2.11. Acceptor

Hereda de `Thread` y tiene como atributo una referencia al `Socket` server de `WeatherServer` además de una referencia a `WeatherDataProtected`, una a `QuitProtected` y un vector de `Receiver*`. Este objeto define el método virtual puro `run()` con un ciclo que acepta clientes mientras que la variable booleana que encapsula `QuitProtected` sea `false`. Con cada cliente que acepta, se crea un hilo `Receiver`, se comienza su ejecución (De esta forma, los pares que resultan del `Map` se van recibiendo paralelamente mientras que se siguen aceptando clientes) y se agrega su referencia al vector atributo.
Cabe aclarar que `QuitProtected` es un booleano protegido por un `Mutex`, consultado por `Acceptor` y modificado por el hilo principal en `WeatherServer`.

2.12. Receiver

Hereda de `Thread` y tiene como atributo un `Socket` peer a través del cual recibe los pares clave valor enviados por el `WeatherClient`. Este objeto define el método virtual puro `run()` con un ciclo que recibe pares hasta el End\n y envía cada uno de ellos a un `Parser` que los interpreta.

2.13. Parser

Este *functor* toma un par recibido por el `Receiver`, lo ingresa a `WeatherDataProtected` con su clave entero día y valor `WeatherValue` que constan de la temperatura y ciudad.

2.14. Reducer

Hereda de `Thread` y tiene como atributo su entero día, una referencia a un vector de `WeatherValue*` y otra a `WeatherResultsProtected`. Este objeto define el método virtual puro `run()` utilizando el `sort` de la librería *algorithm* para obtener la temperatura máxima y un ciclo para las ciudades afectadas. Al finalizar, se guarda el resultado como una cadena con el formato descripto en el enunciado en `WeatherResultsProtected`.
Cabe aclarar que el `sort` se realiza pasándole el *functor* `ValueCmp` que define como se ordenan los `WeatherValue*`, con la mayor temperatura primero y alfabéticamente por ciudad en caso se temperaturas iguales.

2.15. main

Ambos *mains* se desarrollan de forma similar, instanciando el `Socket` cliente o servidor y pasándolo al `WeatherClient`/`WeatherServer` junto a la información correspondiente recibida en el comando. Se ejecuta el cliente/servidor y el programa finaliza devolviendo `EXIT_SUCCESS` (0).

3. Diagramas

3.1. Clases

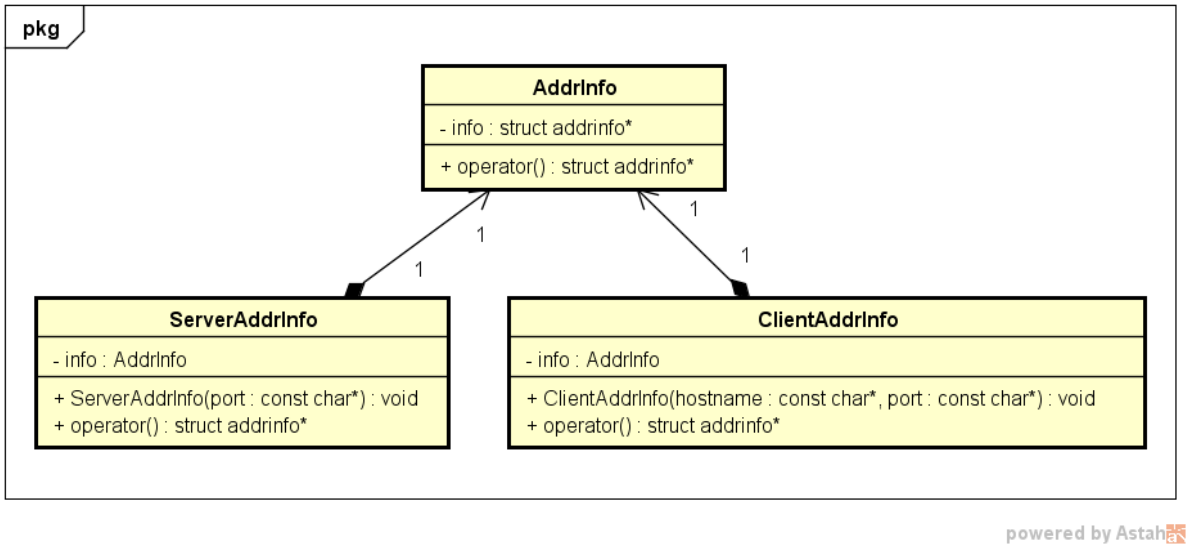

Figura 1: AddrInfo

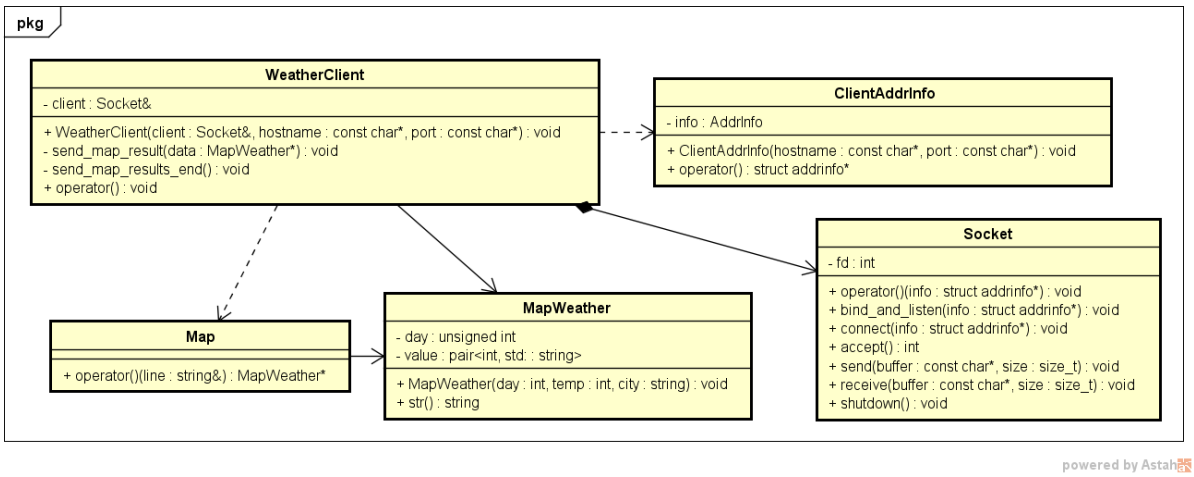

Figura 2: WeatherClient

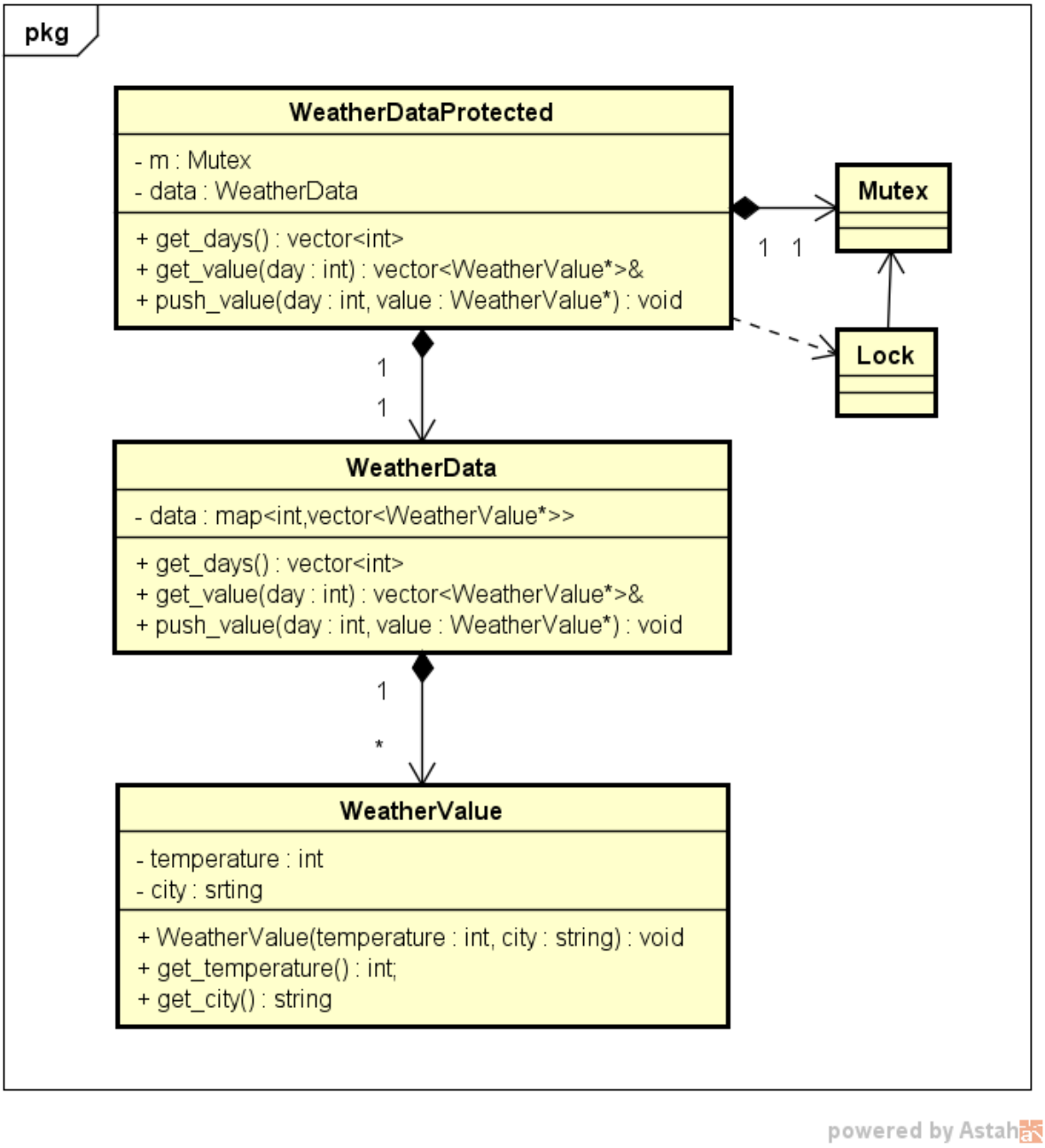

Figura 3: WeatherData

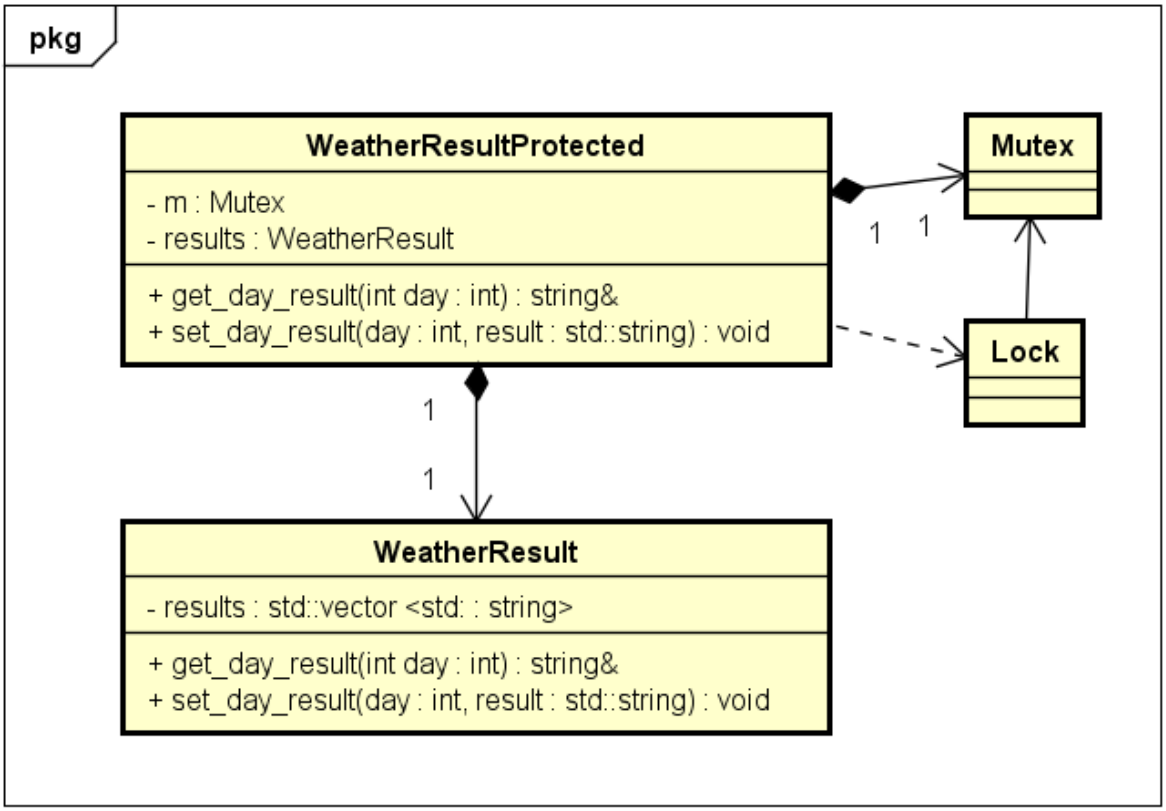

Figura 4: WeatherResults

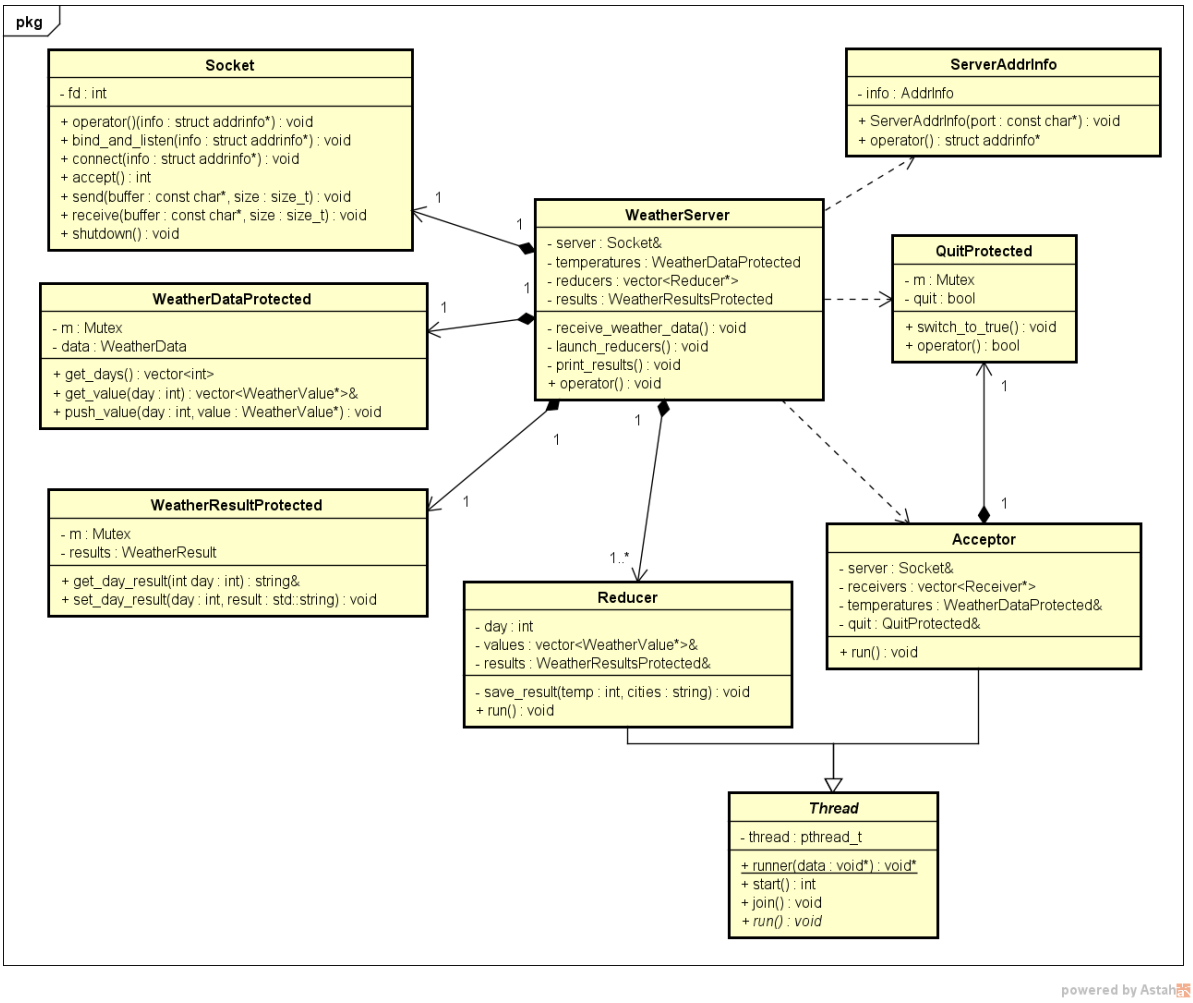

Figura 5: WeatherServer

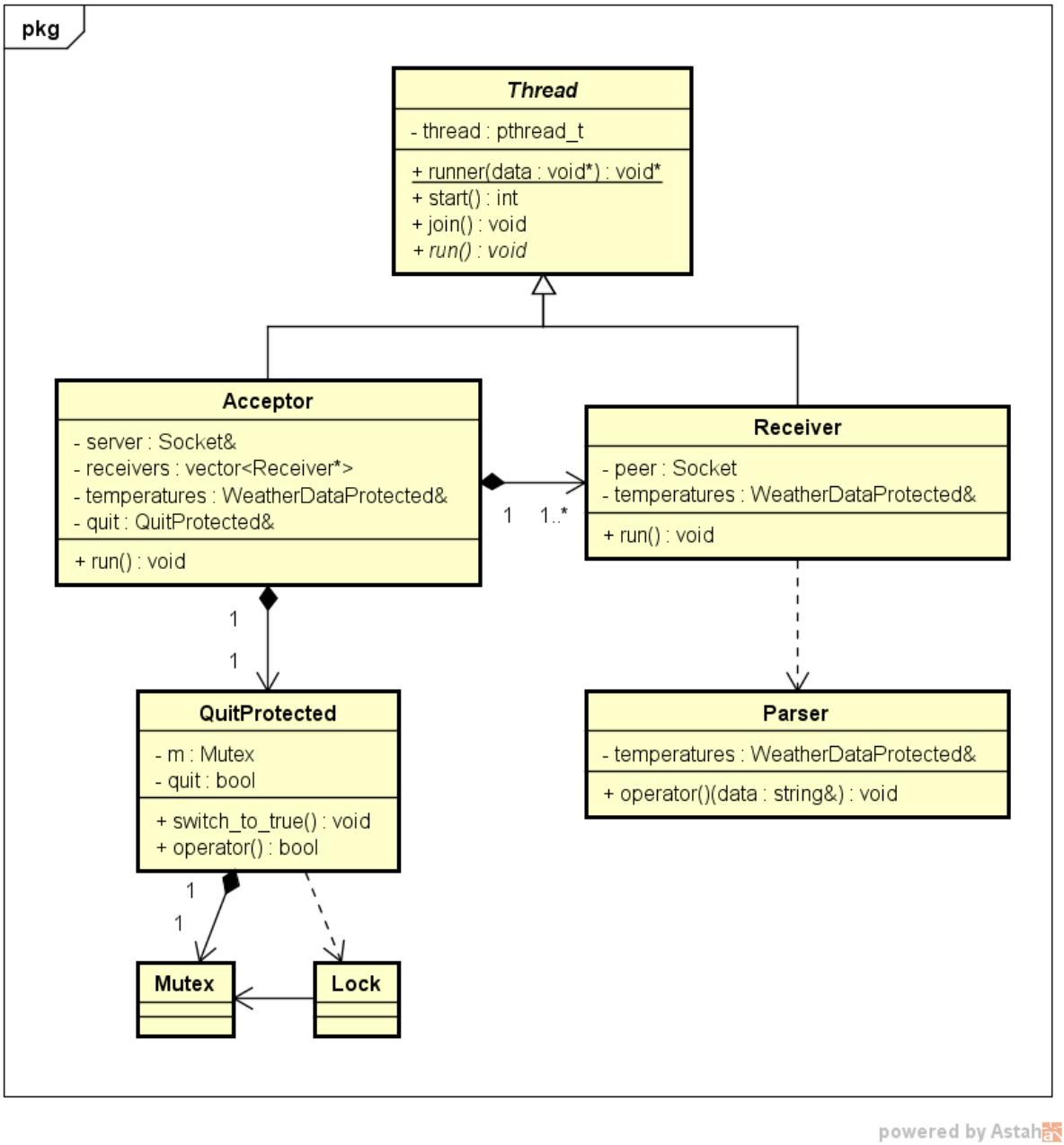

Figura 6: Acceptor