Aggregator Library Tutorial

Introduction

In the world of web services, it is commonplace for a web service to communicate with downstream services to process an API request from its clients. Web service developers try to decouple these actions into as many individual tasks as possible. However, some tasks end up as dependencies for other tasks to execute. Typically, web service developers execute these tasks in a sequential order one after the other. However, in this implementation, the throughput and latency of the web service will take a hit because each task is blocked on the previous task even though there is no dependency between them.

On the other hand, another way of implementation is to execute independent tasks in parallel through creation of threads, and use their result to process the dependent tasks. In this implementation even though it improves the web service's latency and throughput, the developers will have an overhead of thread creation and thread management. A better implementation for the above usecase is to decouple the tasks from their execution and hand it over to an executorservice which takes care of running tasks while honoring their dependencies on other tasks, which is our aim of our library. We have build a task execution library in C++, **Aggregator Library**, which allows web service developers to exploit parallelism with minimal overhead of handling thread creation and thread management at the same time honoring task dependencies. Our C++ Aggregator Library, takes a set of tasks with their dependencies as input, builds a dependency graph for these tasks and executes these tasks as concurrently as possible.

Example Use-case

Let us take an example to illustrate the problem our library tries to solve. In Fig 1, we can see four tasks **A, B, C** and **D** with their dependencies. Task A is independent and can execute first, whereas Tasks B and C depend on A since they use Task A's output. Finally Task D is dependent on outputs of Tasks B and C and hence it has to execute last.

So typically, the common schedule of execution that developers prefer or use is sequential which is shown below Fig 2. In the sequential execution, the main problem is that for web services which are latency critical, will affect their request handling time since developers are not utilizing the independency of Tasks B and C. Fig 3 shows an implementation of these tasks in C++ for their sequential execution.

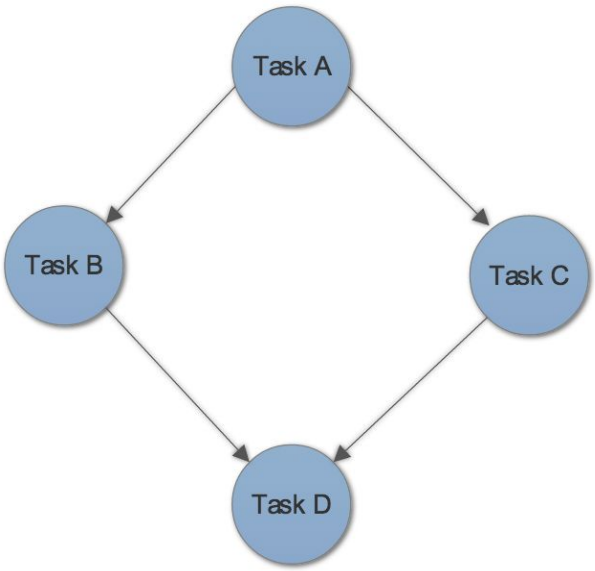

Fig 1 Tasks Dependency Graph

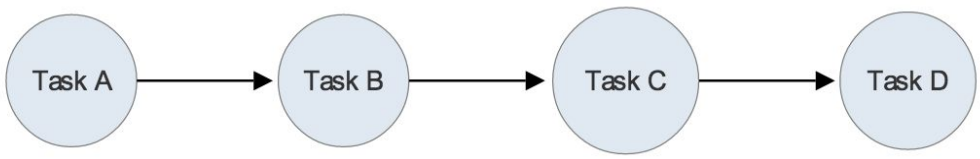

Fig 2 Tasks Execution Schedule

```
int main() {¶
»   A objA; B objB; C objC; D objD;¶
»   objA.run();¶
»   objB.run(objA);¶
»   objC.run(objA);¶
»   objD.run(objB, objC);¶
»   return 0;¶
}¶
```

Fig 3

The ideal way of execution would be to implement Task B and C in parallel by creating two parallel threads say, Thread B and Thread C which execute Task B and Task C respectively. Finally the main thread will wait for the child threads to complete and finally execute Task D. Fig 4 depicts the invoking strategy for parallel execution.

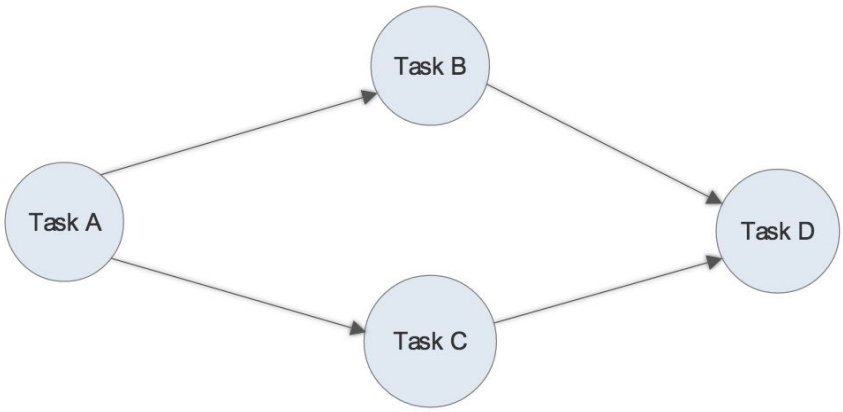

Fig 4

To achieve the above parallel execution of independent tasks, Fig5 describes the C++ implementation which creates threads and executes these tasks.

```
int main() {
    A objA; B objB; C objC; D objD;
    objA.run();

    thread B{objB, &objA};
    thread C{objC, &objA};
    B.join();
    C.join();

    objD.run(objB, objC);
    return 0;
}
```

Fig 5

Aggregator Library

The main problem in the Fig 5 C++ implementation is that, the developers have to deal with thread creation, management, and dependency injection of the output results to dependent tasks. This is the problem, that our aggregator library tries to address and mitigates it, by taking away the burden of thread creation, management and dependency injection from developers. Fig 6 shows the rewritten code using our aggregator library.

```
using Runnable = shared_ptr<Runnable>;
using Aggregator = shared_ptr<Aggregator>;

int main() {
    Aggregator aggregator = AggregatorFactory::newFixedThreadPoolAggregator();
    aggregator->addNodes(Runnable(new A()), Runnable(new B()),
        Runnable(new C()), Runnable(new D()));
    aggregator->execute();
    return 0;
}
```

Fig 6

Runnable Interface

For a client to create a task that can be sent to Aggregator for execution, it has to implement the **Runnable** interface. Fig 7 shows the Runnable interface and its pure virtual methods that the client has to override.

```
class Runnable{
public:
    virtual std::string getLabel() const = 0;
    virtual std::unordered_set<std::string> getDependencies() const = 0;
    virtual void run(std::unordered_map<std::string, std::shared_ptr<Runnable>>) = 0;
    virtual ~Runnable() {}
};
```

Fig 7

The getLabel() method must return the label for the client's task. No tasks should have the same label. In such cases, the aggregator library considers the last task submitted to it and ignores all the remaining tasks which have the same label. It's good practice to name label same as the Task's class name.

The getDependencies() method must return the set of labels of tasks that this current task is dependent on. The task should return the set of existing tasks labels. However, in the case where a label for which a corresponding task doesn't exist the aggregator library ignores the dependency for that task.

Finally, the run() method is the place where the client's task logic goes. The input to the run method gives access to the dependencies of the current task, where it can access their state by performing dynamic_cast if required. Fig 8 shows a task which implements the runnable interface with all the above methods.

```
class B : public Runnable {
private:
    string label{"B"};
    unordered_set<string> dependencies{"A"};
public:
    string getLabel() const {
        return label;
    }

    unordered_set<string> getDependencies() const {
        return dependencies;
    }

    void run(unordered_map<string, shared_ptr<Runnable>> dependencies) {
        cout << "Running node" << label << " with dependencies ";
        for(auto it = dependencies.begin(); it != dependencies.end(); it++) {
            shared_ptr<Runnable> node = it->second;
            cout << node->getLabel() << " ";
        }
        cout << endl;
    }
};
```

Fig 8

Aggregator

Clients need to include the aggregator.h file to access the aggregator library. They can invoke the static Aggregator Factory method newFixedThreadPoolAggregator which creates an Aggregator instance and returns a shared_ptr to it. Fig 9 shows the interface for the Aggregator which are the only methods that the client can use to interact with the Aggregator Library.

```
class Aggregator {
public:
    virtual void addNode(std::shared_ptr<Runnable>) = 0;
    virtual void addNodes(std::shared_ptr<Runnable>, ...) = 0;
    virtual void execute() throw(CyclicDependencyFoundException) = 0;
    virtual ~Aggregator() {}
};
```

Fig 9

The addNode() method adds a single task of client's for execution. The addNodes() method is a variadic function which takes variable number of tasks as input for their execution similar to addNode(). Finally, once all the tasks are added to the Aggregator the clients need to invoke the execute() method which performs execution of all tasks.