A Short Introduction to DAPro

2012 年 10 月 15 日

1 Introduction

DAPro, 全称Distributed Algorithm Fast Prototyping Platform, 旨在为分布式算法的设计、测试以及实现提供一个快速原型平台。DAPro 是基于discrete event simulation (离散事件模拟, 以下简称为DES) [BC84] 理念而设计的。在DES 中，系统的运作是由一系列在离散时间发生的事件所驱动的。DES 系统通常包含以下几个组件(component):

1. Clock，时钟。系统时间是离散的，可以使用初值为0，不断递增推进的整数序列来模拟。时间为DES 其他组件所共享。比如，事件(event) 会带有时间标签，表明其何时应该被调度(schedule)。系统引擎(engine) 会根据被模拟的场景逻辑计算或者判断何时应该产生新的事件，何时应该调度某事件等。

2. Event, 事件。事件是一个DES 系统的核心概念。设计DES 系统，需要明确定义在该系统中哪些行为被视为事件。事件可以具有逻辑上的分类，比如有消息事件(发送消息事件，接收消息事件)，延时事件(随机延时，定时延时等) 和异常事件(节点失效事件)等。

3. EventList, 事件表。事件带有时间标签，所有的事件通常按照时间先后顺序排列在事件表中，以等待系统引擎(engine) 调度执行。该事件表通常使用优先级队列来实现。

4. Engine, 引擎。引擎是DES 系统的控制组件。它负责系统的启动，运行和终止。启动时通常会产生初始事件；系统运行期间不断产生新的事件并调度执行；引擎会判断终止条件是否成立。

设计DAPro 是面向distributed algorithm 的，我们目前采用distributed computing 中的异步消息传递模型(asynchronous message passing model) [AW04]。分布式系统由物理上分布的多个处理器构成，各处理器通过在通信信道上发送消息进行通信。每个信道在两个特定的处理器之间提供一个双向连接。处理器节点与信道组成了该系统的拓扑图。

2 Design

本设计分为两个部分。第一部分为系统模型。第二部分为DES 各组件设计。

2.1 System model part of DAPro

系统拓扑见图1。注意，图中文字与源码中类名 (package *core.impl.messagepassing.topology*) 保持一致。

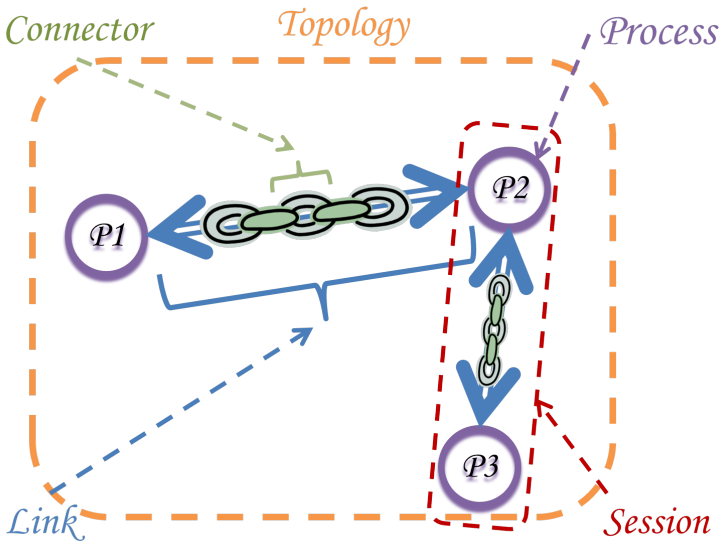

Figure 1: Topology of system

说明如下：

- Topology 采用Singleton 模式。它负责建立、维护、查询系统拓扑。系统拓扑包括Processes 和Session 两部分。Topology 使用id2Process

和pair2Session 两个HashMap 数据结构来表示。

- Session 对象维护了两个通过信道连接了的进程，即fromProcess, toProcess, Link (注：此处的Session 表示单向链接。可以考虑扩展实现双向session)。
- Link 对象表示信道。信道负责传输信息。信道可由不同类的Connector 对象以链表的形式组装而成。在此结构基础上，Connector 负责最底层信息处理。

2.2 DES part of DAPro

本小节重点介绍Event 和Engine 的设计。

2.2.1 Event

Event 的定义包含两个要素，一为Event 的处理时间(triggeringClock), 二为Event 的处理函数(*IEventHanlder* 实例)。在实例化一个Event 对象时，需要确定以上两个参数。*Event* 与*IEventHandler* 均见Package *core.event*。

凡是需要处理Event 的实体(*SimulationObject* 实例)均需要*implements IEventHandler* 接口，并实现其中的*dispatch(Event e)* 抽象方法用以对Event 做任何必要的处理。

IEventCollection 是Eventlist (事件列表) 的抽象实现，*PQEventCollection* 是Eventlist 的优先级队列版本的具体实现。

2.2.2 Engine

Engine 见Package *core.engine*，采用Singleton 模式。Engine 实现了DES 的核心控制功能，其主要逻辑由方法*run* 实现：

- 从Eventlist 中取出第一个事件e。这也是下一个要调度执行的事件。
- 将模拟时钟Clock 推进到事件e 的triggeringClock，表示即将调度该事件。
- 调用事件的*action* 方法将调度任务委托给事件实例的*IEventHandler* 去处理。

Engine 会重复执行以上逻辑，直到：

- 模拟时钟超过了最大允许时间。
- 或，事件列表为空。

在某事件Event 的调度执行过程中，可能会产生新的Event (是否产生Event，以及产生什么类型的Event 由实际场景决定)，这些新产生的Event 需要加入到事件列表中，以便Engine 有机会在上述循环中调度并执行这些事件。这些事件可以通过方法*scheduleEvent* 和*scheduleEventAtOnce* 插入到事件列表中。

3 Simple Test

本节使用一个简单Test 来说明DAPro 的设计与实现。

3.1 Go into asynchronous message passing model

本Test 采用Asynchronous Message Passing Model，其实现集中在Package *core.impl.messagepassing.topology* 和*core.impl.messagepassing.event* 中。

core.impl.messagepassing.topology 中的*Process* 表示进程。*Process* 具有唯一pid，并预定义(作为构造函数的参数) 了其能处理的事件以及相应的事件处理动作(*IProcessAction*)。需要保证的一点是，*Process* 在方法*dispatch(Event e)* 中接收到的Event 类型必须是该进程可以处理的，也就是*event2Action* 数据结构中预定义的。

AsynMessagePassingCom 有两个职责：

1. 一为，代理Process 发送Message。该功能仅仅为将*Message* 对象封装构造成*MessageEvent*，并插入事件列表供Engine 调度。
2. 二为，作为*IEventHandler* 接口的实现者，仅仅负责处理MessageEvent。该处理也很简单，就是根据fromProcess 和toProcess 向Topology 查询到对应的Session，然后转交处理权即可。

3.2 Simple test

该测试用例对应与源码test 文件夹下的Package *core.impl.messagepassing* *MessagePassingTest* 为主测试类。它构造了如下Topology 结构：

- 两个*SimpleProcess* 实例。*SimpleProcess* 可以处理*ProcessStartEvent* 事件和*MessageEvent* 事件，并在运行期间简单地发送一条消息。
- 两个*SimpleProcess* 对象之间建立一条Link。该Link 内仅包含一个Connector，即*FixedDelayConnector*。该Connector 会使得当前事件产生延时并重新插入到事件列表等待再次调度。

Engine 产生*ProcessStartEvent* 作为初始事件，插入事件列表，然后启动Engine 运行。在该测试系统运行过程中，进程之间会发送一条Message (从p1 到p2)，该Message 在Link 中会由*FixedDelayConnector* 进行延时处理，并最终到达进程p2。进程p2 接受到该MessageEvent 消息，调用事先注册的*MessageAction* 对象来处理该消息。

4 Future Work

下面列举几个可做的改进和补充：

1. Topology 允许修改。实现双向Session 类型。实现多种类Connector。
2. 代码重构。
3. 添加模拟数据统计功能。比如，为完成某算法所需的消息数目。
4. 考虑采用Java 反射机制简化“事件与事件处理” 之间的关联
5. 实现fault Event 类型。比如Process 失效，Link 失效等。
6. 设计、实现更复杂的场景。
7. 考虑Shared Memory Model 的设计和实现。
8. XML 设计，GUI 设计。

References

[AW04] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*, volume 19. Wiley-Interscience, 2004.

[BC84] J. Banks and J.S. Carson. Discrete event system simulation. 1984.