Паттерн «Многоагентная система»

Кушнир П. М., октябрь 2011
версия для публикации 05.10.2012

Проблема.

Необходимо обеспечить корректную работу системы разнородных асинхронных компонент (сетевые службы, объекты БД, пользовательские интерфейсы).

Предпосылки.

Асинхронность и параллелизм

В некоторых параллельных системах программирования передача данных между компонентами скрыта от программиста, тогда как в других она должна указываться явно. Явные взаимодействия могут быть разделены на два типа:

- Взаимодействие через разделяемую память (например, в Java или C#). Данный вид параллельного программирования обычно требует какой-то формы захвата управления (мьютексы, семафоры, мониторы) для координации потоков между собой
- Взаимодействие с помощью передачи сообщений (например, в Erlang или occam). Обмен сообщениями может происходить асинхронно, либо с использованием метода «рандеву», при котором отправитель блокирован до тех пор, пока его сообщение не будет доставлено. Асинхронная передача сообщений может быть надёжной (с гарантией доставки) либо ненадёжной.

Параллельные системы, основанные на обмене сообщениями, зачастую более просты для понимания, чем системы с разделяемой памятью, и обычно рассматриваются как более совершенный метод параллельного программирования.

Многоагентная система

Многоагентная система (МАС, англ. Multi-agent system) — это система, образованная несколькими взаимодействующими интеллектуальными агентами. Многоагентные системы могут быть использованы для решения таких проблем, которые сложно или невозможно решить с помощью одного агента или монолитной системы.

В многоагентной системе агенты имеют несколько важных характеристик:

- *Автономность*: агенты, хотя бы частично, независимы
- *Ограниченность представления*: ни у одного из агентов нет представления о всей системе, или система слишком сложна, чтобы знание о ней имело практическое применение для агента.
- *Децентрализация*: нет агентов, управляющих всей системой

Обычно в многоагентных системах исследуются программные агенты. Тем не менее,

составляющими мультиагентной системы могут также быть роботы, люди или команды людей. В данном случае, прослойкой между агентом-человеком является командный агент, связаный с пользовательским интерфейсом.

Агент

В компьютерной науке интеллектуальный агент — программа, самостоятельно выполняющая задание, указанное пользователем компьютера. Агент воспринимает информацию от других агентов с помощью перцепторов и может обладать возможностью взаимодействовать с другими агентами с помощью действий. Особенностью агента является ограниченное количество известных средств взаимодействия с другими агентами. Множество действий агента, обусловленное информацией от перцепторов агента назовём поведением. В зависимости от назначения, агент может иметь или не иметь внутреннего состояния.

Среда

Среду можно определить как сущность, которая обладает способностью изменять состояние перцепторов агента предоставляет агентам возможность выполнения действий. Важной особенностью среды является конечная скорость распространения изменений, вызваных действиями агентов. Это означает появление некоего промежутка времени между действием одного агента и изменением состояния перцептора другого агента. Таким образом, включение в систему временной характеристики позволит описывать процессы взаимодействия агентов, которые по тем или иным причинам не могут быть осуществлены мгновенно.

Решение.

Выбор средства разработки.

В качестве основного средства разработки будет использована среда *BlackBox Component Builder*, в качестве языка — *Component Pascal*. Каркас среды обеспечивает необходимые средства работы с загружаемыми модулями и асинхронными процедурами (отложенные действия Services.Action).

Также необходимо инсталлировать в среду подсистему Abf, которая предоставляет функциональность межмодульной шины сообщений (AbfBus), и реализует паттерн «Объект-сообщение».

В среде BlackBox *модуль* является единицей компиляции и исполнения. Механизм загрузки обеспечивает наличие единственного экземпляра модуля в памяти среды. Также, модуль обеспечивает различные уровни доступа (*полный*, *только-чтение*, *скрытый*) ко включённым в него классам, объектам и процедурам. Множество доступных для чтения или записи элементов модуля определяет его *интерфейс*.

Средства мета-программирования и рефлексии позволяют организовать механизм работы с объектами произвольного модуля, на основе интерфейса модуля. На основе этой функциональности подсистема Abf предоставляет разработчику средства передачи произвольного объекта-сообщения в специальную процедуру-обработчик, называемую хэндлером. Передачу объекта в данную процедуру будем называть отсылкой сообщения. Реализация модуля AbfBus предоставляет средства адресной и широковещательной рассылки объектов-сообщений.

Асинхронные процедуры в среде BlackBox являются методами реализации абстрактного класс Services.Action и могут быть исполнены средой по истечению запланированного промежутка времени. Варианты реализации механизма отложенных действий предполагают изолированность от методов реализации параллельных алгоритмов, предоставляемых нижележащей операционной системой. В данной реализации среды BlackBox подобные средства не задействованы вовсе.

Концепция агентов в среде BlackBox.

Проанализировав характеристики ранее описаных сущностей можно определить необходимые сущности для реализации асинхронной многоагентной системы в среде BlackBox.

Элемент модели	Сущность среды BlackBox
Среда	Шина сообщений или вспомогательный модуль по работе с шиной.
Агент	Модуль
Перцептор	Процедура-хэндлер
Информация перцептора	Объект-сообщение
Поведение	Отложенное действие
Действие агента	Отправка объекта-сообщения в шину

Обусловим, что реализацией агента является модуль, который обладает перцептором и удовлетворяет последующим уточнениям реализации.

То есть: *всякий агент — модуль, но не всякий модуль — агент.*

Также необходимо оговорить, что для корректной работы реализации модели принимается, что само взаимодействие среды с перцептором и агента со средой является мгновенным. Так, для текущей реализации BlackBox это означает дополнительную договорённость о запрещении выполнения дополнительных «тяжёлых» операций в хэндлере, кроме как выбор нужного поведения. Если подобный выбор поведения сам является «тяжёлой» операцией, желательно выразить его так же в виде поведения.

Пример реализации.

Реализуем модуль-среду.

```
DEFINITION MasEnv;

IMPORT AbfBus;

TYPE
	Message = ABSTRACT RECORD (AbfBus.Message) END;

PROCEDURE Do (VAR msg: Message);
PROCEDURE Init (IN modName: ARRAY OF CHAR);

END MasEnv.
```

Мы определили абстрактный тип сообщения, этот тип станет основой для всех реализаций конкретных сообщений агентов. Также мы определили процедуру, которая предоставит агентам возможность выполнить действие (т. е. инициировать рассылку объекта-сообщения остальным агентам).

Реализуем несколько произвольных агентов, которые будут обладать различным поведением, набором типов сообщений для перцепторов, и набором типов сообщений действий.

```
DEFINITION MasNet;

IMPORT MasEnv, AbfBus;

CONST
	error = -1;
	offline = 0;
	ok = 0;
	online = 1;

TYPE
	ConnectMsg = RECORD (MasEnv.Message) END;

	ConnectedMsg = LIMITED RECORD (MasEnv.Message) END;

	DataInMsg = LIMITED RECORD (MasEnv.Message)
		data-: ANYPTR
	END;

	DataOutMsg = LIMITED RECORD (MasEnv.Message)
		data-: ANYPTR
	END;

	DisconnectMsg = RECORD (MasEnv.Message) END;

	DisconnectedMsg = LIMITED RECORD (MasEnv.Message) END;

	ErrorMsg = LIMITED RECORD (MasEnv.Message)
		fatal-: BOOLEAN
	END;

	GetStateMsg = RECORD (MasEnv.Message)
		state-: INTEGER
	END;

	SendDataMsg = RECORD (MasEnv.Message)
		data: ANYPTR;
		res-: INTEGER
	END;

PROCEDURE HandleBusMsg (VAR msg: AbfBus.Message);
PROCEDURE Init;

END MasNet.
```

Был реализован макет агента, поддерживающего асинхронное сетевое соединение, этот агент предоставляет возможность управления соединением, а также возможность получать данные и отправлять данные с помощью установленного соединения. Неотъемлимой частью модуля-агента является хэндлер сообщений HandleBusMsg.

Ключевой особенностью агента является схема работы Запрос — Ответ — Соглашение. Согласно этой схеме мы описываем сообщения-сигналы перцептора. Возможности языка Component Pascal позволяют гибко регулировать свойства типов сообщений. Так, для типов-сигналов мы устанавливаем полный доступ к полям объекта-

сообщения, как например поле data в сообщении SendDataMsg. А для сообщений-действий мы устанавливаем тип LIMITED что позволит нам запретить размещение объектов-сообщений в сторонних модулях, а следовательно, это обеспечивает однозначную связь агента-модуля и его действий.

Для полноты картины рассмотрим сразу второй макет агента.

```
DEFINITION MasCmds;

        IMPORT AbfBus;

        PROCEDURE HandleBusMsg (VAR msg: AbfBus.Message);
        PROCEDURE Init;
        PROCEDURE Send;
        PROCEDURE Start;
        PROCEDURE State;
        PROCEDURE Stop;

END MasCmds.
```

Этот агент предоставляет собой агент взаимодействия с пользователем. Набор команд этого модуля описан в формате процедур без параметров. Этот формат является форматом для связывания команд модуля с визуальным компонентом «Коммандер» среды BlackBox, который реагирует на нажатие мыши и выполняет заданную команду. Таким образом, в данном примере, только пользователь решает, в какой момент времени будет вызвана та или иная команда, следовательно, это асинхронный агент.

Изучив исходный код, размещёный в приложении к данной статье, можно увидеть, что команды пользователя однозначно связаны с сообщениями-сигналами, описанными в сетевом агенте.

При загрузке модуля MasCmds происходит загрузка модулей MasEnv (среда) и MasNet (сетевой агент).

Затем, вызывается скрытая процедура MasCmds.Init которая вызывает команду инициализации модуля в среде MasEnv.Init (имя_модуля). После этого агенты переходят в режим ожидания сигналов.

При вызове MasEnv.Do(msg) происходит рассылка сообщений по хендлерам зарегистрированых модулей.

В свою очередь, хэндлер сетевого агента реагирует на эти сообщения. Благодаря средствам языка Component Pascal возможно фильтрация входящих сообщений на уровне типа сообщения, и агент может ограничивать множество обрабатываемых сообщений.

Второй ключевой момент, это мгновенная реакция на сообщение. Так, при получении сообщения ConnectMsg агент регистрирует экземпляр отложенного действия типа ConnectAction которое является реализацией поведения агента. Так как сам процесс регистрации является синхронным, он не вызывает пауз и управление тут же возвращается к BlackBox. По прошествии некоторого времени BlackBox производит вызов метода ConnectAction.Do которое в зависимости от состояния соединения предпринимает попытку физического соединения, либо инициирует **действие** по сообщению результата подключения (ConnectedMsg, ErrorMsg).

В целях упрощения реализация соединения выполнена в виде периодического действия, которое имитирует поведение реального соединения, например получение данных

или ошибки на линии.

Также можно рассмотреть действие по отправке пакета данных. Оно отличается тем, что в типе сообщения SendDataMsg описано поле res, которое даёт возможность узнать результат мгновенного действия по размещению пакета данных. Сделано по причине изолированности агентов. Так как агенты не обязаны знать полную информацию о системе и других агентах, может случиться так, что сторонний агент предпримет действие по отправке данных при отключенном соединении. Агент сетевого соединения должен корректно обработать данную ситуацию и сообщить незамедлительно, так как предпринимать какие-то действия не имеет смысла.

Таким же образом обрабатывается сигнал запроса состояния соединения GetStateMsg.

В ином случае, при правильном состоянии подключения, сетевой агент принимает пакет данных в очередь отправки, а соединение после попытки отправки данных инициирует действие уведомления о результатах отправки (DataOutMsg, ErrorMsg).

Ранее были рассмотрены случаи ответа агента на запрос от другого агента. В результате каждого запроса агенты приходили к соглашению, которое заключалось в выполнении неких действий и возврате результата этих действий.

Однако, есть ещё один вариант собственного поведения сетевого агента. В процессе работы соединения может происходить получение даннных. Тогда агент должен уведомить остальных агентов о получении данных. Поэтому он предпринимает попытку действия по отправке DataInMsg, которое могут обработать сторонние агенты, как это сделано в хэндлере агента пользователя MasCmds.

Таким же образом осуществляется уведомление агентов об обрыве.

Результаты.

Были рассмотрены основные принципы построения многоагентных систем, проведён анализ доступных средств разработки и произведён макет многоагентной системы с асинхронным взаимодействием агентов.

Преимуществами агентной системы можно назвать гибкость поведения агентов, безопасность их внутреннего состояния, интеллектуальность принятия решений по реакции на тот или иной сигнал в зависимости от внутреннего состояния.

Недостатками данной реализации агентной системы является необходимость более детальной проработки структуры системы в целом, необходимость контроля за критичными к асинхронной работе, сторонними компонентами, и некоторые другие известные «тонкие» моменты многопоточного программирования.

Также наследуется проблема широковещания в модульной системе, которая может выражаться в холостой рассылке сообщений модулям, которые в них не нуждаются. Но эта проблема решаема и некритична для данного макета системы.

Приложение 1. Исходные коды.

```
MODULE MasEnv;

	IMPORT AbfBus, Dialog;

	TYPE
		Message* = ABSTRACT RECORD (AbfBus.Message) END;

	PROCEDURE Do* (VAR msg: Message);
		VAR res: INTEGER;
	BEGIN
		AbfBus.Broadcast(msg, res);
	END Do;

	PROCEDURE Init*(IN modName: ARRAY OF CHAR);
		VAR res: INTEGER;
	BEGIN
		Dialog.Call(modName+'.Init', 'ошибка инициализации агента '+modName$, res);
	END Init;

END MasEnv.
```

```
MODULE MasNet;

	IMPORT Services, MasEnv, AbfBus, Log;

	CONST
		online* = 1;
		offline* = 0;
		maxTryNum = 5;
		error* = -1;
		ok* = 0;

	TYPE
		DataInMsg* = LIMITED RECORD (MasEnv.Message)
			data-: ANYPTR;
		END;

		DataOutMsg* = LIMITED RECORD (MasEnv.Message)
			data-: ANYPTR;
		END;

		ConnectedMsg* = LIMITED RECORD (MasEnv.Message) END;

		DisconnectedMsg* = LIMITED RECORD (MasEnv.Message) END;

		ErrorMsg* = LIMITED RECORD (MasEnv.Message)
			fatal-: BOOLEAN;
		END;

		GetStateMsg* = RECORD (MasEnv.Message)
			state-: INTEGER;
		END;

		SendDataMsg* = RECORD (MasEnv.Message)
			data*: ANYPTR;
			res-: INTEGER;
		END;

		ConnectMsg* = RECORD (MasEnv.Message) END;

		DisconnectMsg* = RECORD (MasEnv.Message) END;

		Connection = POINTER TO RECORD (Services.Action)
			t: LONGINT;
			state: INTEGER;
			buf: POINTER TO RECORD END;
		END;

		ConnectAction = POINTER TO RECORD (Services.Action)
			connection: Connection;
			tryNum: INTEGER;
		END;

		DisconnectAction = POINTER TO RECORD (Services.Action)
			connection: Connection;
		END;

		State = POINTER TO RECORD
			connection: Connection;
		END;

	VAR
		state: State;

	PROCEDURE (a: Connection) Do;
		VAR im: DataInMsg; om: DataOutMsg; em: ErrorMsg;
	BEGIN
		CASE a.state OF
			|online:
				IF (Services.Ticks() - a.t) > 1000 THEN
```

```
                                MasEnv.Do(im);
                                a.t:=Services.Ticks();
                        END;
                        IF (a.buf#NIL) THEN
                                om.data:=a.buf;
                                a.buf:=NIL;
                                IF ODD(a.t DIV 1000) THEN
                                        MasEnv.Do(om);
                                ELSE
                                        em.fatal:=FALSE;
                                        MasEnv.Do(em);
                                END;
                        END;
                        Services.DoLater(a, Services.now);
                |offline:
                        em.fatal:=TRUE;
                        MasEnv.Do(em);
        ELSE HALT(100) END;
END Do;

PROCEDURE (a: Connection) Init, NEW;
BEGIN
        a.state:=offline;
        a.t:=Services.Ticks();
END Init;

PROCEDURE (a: Connection) State(): INTEGER, NEW;
BEGIN
        RETURN a.state;
END State;

PROCEDURE (a: Connection) Connect, NEW;
BEGIN
        a.state:=online;
END Connect;

PROCEDURE (a: Connection) Disconnect, NEW;
BEGIN
        a.state:=offline;
END Disconnect;

PROCEDURE (a: Connection) Send(data: ANYPTR; OUT res: INTEGER), NEW;
BEGIN
        IF a.state = online THEN
                NEW(a.buf);
                res:=ok;
        ELSE res:=error END;
END Send;

PROCEDURE (a: ConnectAction) Do;
        VAR em: ErrorMsg; cm: ConnectedMsg;
BEGIN
        IF (a.connection.State() = offline) & (a.tryNum<maxTryNum) THEN
                a.connection.Connect;
                INC(a.tryNum);
                Services.DoLater(a, Services.Ticks()+1000);
        ELSIF (a.connection.State() = offline) & (a.tryNum=maxTryNum) THEN
                em.fatal:=TRUE;
                MasEnv.Do(em);
        ELSIF (a.connection.State()=online) THEN
                MasEnv.Do(cm);
                Services.DoLater(a.connection, Services.now);
        END;
END Do;

PROCEDURE (a: DisconnectAction) Do;
        VAR dm: DisconnectedMsg;
BEGIN
        a.connection.Disconnect();
        Services.RemoveAction(a.connection);
```

```
		MasEnv.Do(dm);
	END Do;

	PROCEDURE (s: State) Connect, NEW;
		VAR ca: ConnectAction;
	BEGIN
		IF s.connection = NIL THEN
			NEW(s.connection); s.connection.Init;
			NEW(ca); ca.connection:=s.connection; ca.tryNum:=0;
			Services.DoLater(ca, Services.now);
		END;
	END Connect;

	PROCEDURE (s: State) Disconnect, NEW;
		VAR da: DisconnectAction;
	BEGIN
		IF s.connection # NIL THEN
			NEW(da); da.connection:=s.connection;
			Services.DoLater(da, Services.now);
			s.connection:=NIL;
		END;
	END Disconnect;

	PROCEDURE HandleBusMsg* (VAR msg: AbfBus.Message);
	BEGIN
		WITH
			|msg: ConnectMsg DO state.Connect;
			|msg: DisconnectMsg DO state.Disconnect;
			|msg: SendDataMsg DO
				IF (state.connection#NIL) & (msg.data#NIL) THEN
					state.connection.Send(msg.data, msg.res);
				ELSE msg.res:=error END;
			|msg: GetStateMsg DO
				IF (state.connection#NIL) THEN
					msg.state:=state.connection.State();
				ELSE
					msg.state:=offline;
				END;
		ELSE END;
	END HandleBusMsg;

	PROCEDURE Init*;
	BEGIN
		Log.String('инициализация сетевого агента'); Log.Ln;
	END Init;

BEGIN
	NEW(state)
CLOSE
	state.Disconnect;
END MasNet.
```

```
MODULE MasCmds;

	IMPORT MasEnv, MasNet, AbfBus, Log;

	PROCEDURE Msg(IN s: ARRAY OF CHAR);
	BEGIN
		Log.String(s); Log.Ln;
	END Msg;

	PROCEDURE Start*;
		VAR cm: MasNet.ConnectMsg;
	BEGIN
		MasEnv.Do(cm);
		Msg('попытка установить соединение');
	END Start;

	PROCEDURE Send*;
		VAR sm: MasNet.SendDataMsg; data: POINTER TO RECORD END;
	BEGIN
		Msg('попытка отправить данные');
		NEW(data);
		sm.data:=data;
		MasEnv.Do(sm);
		IF sm.res#MasNet.ok THEN
			Msg('неустранимая ошибка соединения, данные не отправлены')
		END;
	END Send;

	PROCEDURE State*;
		VAR sm: MasNet.GetStateMsg;
	BEGIN
		Log.String('состояние соединения: ');
		MasEnv.Do(sm);
		IF sm.state = MasNet.online THEN
			Msg('активно');
		ELSE Msg('неактивно') END;
	END State;

	PROCEDURE Stop*;
		VAR dm: MasNet.DisconnectMsg;
	BEGIN
		Msg('попытка закрыть соединение');
		MasEnv.Do(dm);
	END Stop;

	PROCEDURE HandleBusMsg* (VAR msg: AbfBus.Message);
	BEGIN
		WITH
			|msg: MasNet.ConnectedMsg DO Msg('соединение установлено');
			|msg: MasNet.DisconnectedMsg DO Msg('соединение разорвано');
			|msg: MasNet.ErrorMsg DO
				Msg('ошибка соединения ');
				IF msg.fatal THEN
					Msg('соединение прервано из-за ошибки');
				END;
			|msg: MasNet.DataInMsg DO Log.String('.');
			|msg: MasNet.DataOutMsg DO Msg('пакет данных отправлен');
		ELSE END;
	END HandleBusMsg;

	PROCEDURE Init*;
	BEGIN
		Msg('старт многоагентной системы');
		MasEnv.Init('MasNet');
	END Init;

BEGIN

END MasCmds.
```

Tool for: Mas Demo
Created: 30 сентября 2011 г., 19:57:55

To compile:

(!) DevCompiler.CompileThis
MasEnv MasNet MasCmds

To unload:

(!) DevDebug.UnloadThis
MasCmds MasNet MasEnv

(!) MasCmds.Init выполнить первой
(!) MasCmds.Start выполнить второй
(!) MasCmds.Send выполнять N раз
(!) MasCmds.State выполнять N раз
(!) MasCmds.Stop выполнить в конце