

银行核心业务系统性能测试初探

一、银行核心业务系统的业务介绍

(一)、银行的类型

我国银行体系由三部分构成：即中央银行、政策性银行和商业银行。中国人民银行为中央银行；国家开发银行、中国农业发展银行和中国进出口银行是政策性银行；商业银行分为国有独资商业银行、股份制商业银行、城市商业银行、农村信用社和境内外资银行。我们此次讨论的银行指的是第三种类型即商业银行。

(二)、银行业务的类型

银行业务分类有多种，按业务资金来源的不同，商业银行业务可分为负债业务、资产业务以及中间业务。

负债类型：存款类、外借款类和银行资本类

资产业务：主要包括发放贷款、投资业务和其他资产业务

中间业务：各种托收托付、汇兑、代理等等

从测试的角度来说，按照日常经营的业务频繁程度，银行最主要的业务是存取款业务和贷款收发业务，次之的是每日的换班扎帐和日终结帐，最后的是利息结算和年终结算这类周期性的结算业务。

(三)、银行核心系统性能测试场景测试模型设计

对核心系统的测试，无论是功能测试还是性能测试都是具有相当难度的。目前很多银行都是新旧两套系统并行运行，通过柜员的双倍劳动付出来实现新旧系统的平稳过渡。那么我们能不能在目前这种测试模式上有所改变和突破呢。能否利用工具实现这样大规模用户量的测试呢。银行核心系统的开发成本和报价都是非常高昂，这样的项目报价往往达千万级别甚至亿元级别，同样这样的

项目的测试一样可以有可观的报价。我们如何规划这样的测试项目呢，如何介入呢？这样的项目可以大致分为三个阶段：

第一阶段软件正式上线前。

可以通过测试工具模拟大业务量数据，确保系统的主要功能能够满足系统设计时对性能的要求。

第二阶段软件试运行期间。

通过对系统性能的监控，检查业务系统的在实际工作环境中资源使用是否合理。

第三阶段软件正式运行后。

通过对系统性能的监控，验证前两个阶段的测试结果。并根据正式运行后的实际状况，提出性能调优建议。

最重要的阶段无疑是第一阶段，能在第一阶段发现问题，解决问题，才能减少系统的风险，减少项目的修改代价。（第二、第三阶段更侧重于监控）

对于这样的测试项目，我们将不可避免的面对下面的一系列问题

测试哪些交易？

交易怎么组合？

达到什么目标？

 初始化多少基础数据？

准备多少交易数据？

怎么准备？

 监控什么？

怎么监控？

 有什么风险？

怎么屏蔽？

做什么测试？负载测试？压力测试？容量测试？配置测试？稳定性测试？

被测环境怎么部署？

负载环境怎么部署？

监控环境怎么部署？

1、测试点：

结合银行日常的业务情况，测试点应该包括个人存款、个人取款、对公存款、对公取款、个人贷款、对公贷款、同城票据交换、汇兑等日常业务，还应该包含诸如换班扎帐、日终结帐、月报、季报、结息和年终结算等数据处理业务。

（当然很多银行的结息和年终结算不部署在核心业务系统中）。

2、测试场景

（1）、日常营业场景模拟

- 在线测试：用户量可以通过银行开户的客户数量度量，交易的吞吐量可以通过银行完成的业务数量算出。并结合换班扎帐和日终结帐的操作。
- 并发测试：

（2）、结算业务场景模拟

银行的计算业务，例如结息、月报、季报和年度结算这类业务的用户数量可以通过机构数量来计算，对于系统来说主要关注的侧重点是这类操作对于日常营业场景的影响以及这类操作的资源占用和时间响应。（当然结算类的业务一般安排在晚上执行或者单独系统来处理）

二、银行核心业务系统的架构介绍

在银行业的分布式系统中以交易中间件为核心框架的三层客户机/服务器模式 (C/S/S) 是绝对的主流架构。中间件在银行、电信、金融等大规模关键事务领域中的整合各种异构平台、保证交易完整性等方面表现出了超强的能力。而这个交易中间件的市场份额以 Tuxedo 为最主要。所以说要开展金银行业核心业务系统的性能测试是绕不过基于 Tuxedo 的三层客户机/服务器结构的系统架构研究的。本文将对如何开展对此架构的系统进行性能测试进行了初探。

(一)、什么是 tuxedo

了解 tuxedo 之前先了解中间件的概念。中间件 (Middleware) 和操作系统、数据库是我们常说的三大基础软件。字面理解它其实就是处于平台 (操作系统和硬件设备) 与用户应用软件之间的中间层结构。它管理计算机的资源 and 通讯, 是一个相对独立的系统软件和服务程序。

种类	作用	典型产品
消息中间件	适用于任何需要进行网络通信的系统, 负责建立网络通信的通道, 进行数据或文件发送。消息中间件的一个重要作用是可以实现跨平台操作, 为不同操作系统上的应用软件集成提供服务。	ibm mqseries tonglink/q
交易中间件	适用于联机交易处理系统, 主要功能是管理分布于不同计算机上的数据的一致性, 保障系统处理能力的效率与均衡负载。交易中间件所遵循的主要标准是 x/open dtp 模型。	ibm cic bea tuxedo tongeasy
对象中间件	基于 corba 标准的构件框架, 相当于软总线, 能使不同厂家的软件交互访问, 为软件用户及开发者提供一种即插即用的互操作性, 就像现	iona orbix borland visibroker

	在使用集成块和扩展板装配计算机一样。	ibm componentbroker tongbroker
应用服务器	用来构造 internet/intranet 应用和其它分布式构件应用，是企业实施电子商务的基础设施。应用服务器一般是基于 j2ee 工业标准的。	ibm websphere bea weblogic tongweb
安全中间件	以公钥基础设施 (pki) 为核心的、建立在一系列相关国际安全标准之上的一个开放式应用开发平台，向上为应用系统提供开发接口，向下提供统一的密码算法接口及各种 ic 卡、安全芯片等设备的驱动接口。	entrust entrust tongsec
应用集成服务器	把 workflow 和应用开发技术如消息及分布式构件结合在一起，使处理能方便自动地和构件、script 应用、工作流行为结合在一起，同时集成文档和电子邮件。	lss flowman ibm flowmark vitria businessagiliti

Tuxedo 就是交易中间件。(Transaction for UNIX has been Extended for Distributed Operation, 即被分布式操作扩展之后的 UNIX 事务系统)。它介于客户机与服务器之间，也就是 C/S/S 结构的中间层，它的作用主要是解决传统 C/S 结构的局限性。它可以为构建大规模的分布式 C/S 应用程序提供了事务、通信、安全、内存管理、负载均衡和容错恢复等基础服务。通过 tuxedo 中间件的业务逻辑扩展来实现异构和分布式系统的快速开发部署和快捷调整联机事务处理类型的大型复杂应用。而不用象传统 C/S 结构那样逐个 Client 端去更新和部署新程序。运行于服务器端的事务管理器是整个 tuxedo 结构的关键主件，它是是 tuxedo 服务的核心，它负责提供诸如名字服务、数据路由、负载均衡、事务管理和安全性管理等服务。

(二)、典型的基于 tuxedo 架构的商业银行拓扑

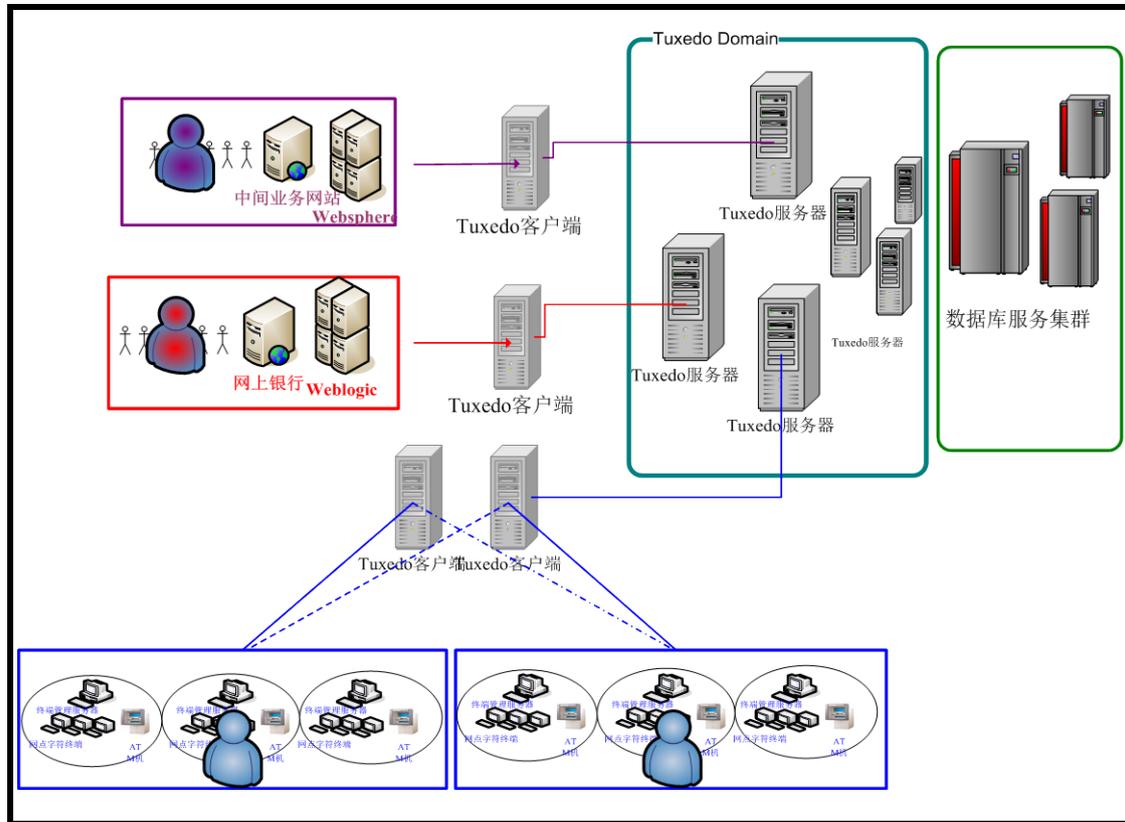


图 1

如上图所示：

上图是一个商业银行的典型拓扑。有柜台综合业务系统系统、网上银行系统和一些中间业务系统（当然中间业务也可以在网上银行和网点完成）。

柜台综合业务系统是 C/S/S 结构，这是整个银行的核心系统。Client 端指的是 Tuxedo 客户端以及其汇聚的终端设备，中间层是 Tuxedo 的服务层，后端对应于数据库服务器。

网上银行和中间业务系统是典型的 BS 结构，但是由于要整合进银行的核心系统，这个时候 tuxedo 就显示出其融合异构系统的超强能力。Tuxedo 通过 connector 来实现与 weblogic 和 webshpere 的应用连接。（WEBLOGIC 与 TUXEDO 的互连有两中方式，通过 JOLT 或通过 WTC (WEBLOGIC TUXEDO CONNECTOR)。它们都是 BEA 的产品，WTC 不仅能让 WEBLOGIC 调用 TUXEDO 中的 SERVICE，而且能让 TUXEDO 调用 WEBLOGIC 中的 EJB；而 JOLT 只能让 WEBLOGIC 调用 TUXEDO。但 JOLT

可以使 TUXEDO 与 WEBSPERE 等其他应用互连)

(三)、架构特点

上述结构图可以看出，银行核心系统如果是基于 Tuxedo 的中间件，那么可以不夸张的说每一笔银行的交易都离不开 Tuxedo。可以说银行是 Tuxedo 完成了银行的业务逻辑。

一个完整的交易应该是最初由 Tuxedo 客户端提起(tpcall), 调用 Tuxedo 服务器的相关 Service, 实现对数据库的操作, 然后 Tuxedo 服务器将结果返回给 Tuxedo 客户端, 客户端再将相关结果消息分发到对应的终端设备, 返回结果的这个过程称为 tpreturn。

三、银行核心业务系统测试重点

银行核心业务系统的测试重点显而易见就是对 Tuxedo 服务器中相关 Service (这些服务对应着 C 或者 C++ 编写的函数) 的测试。不同的业务有不同的 Service 或者也可以叫核心业务处理函数。

(一)、tuxedo 的工作原理

Tuxedo 可以有效地整合企业异构 C/S 系统，实现大规模的关键业务处理和分布式事务管理，从而为企业提供一个可靠的、高性能的、易维护的三层分布式计算机环境。下图展示了一个基本 Tuxedo 系统的组成和工作原理。

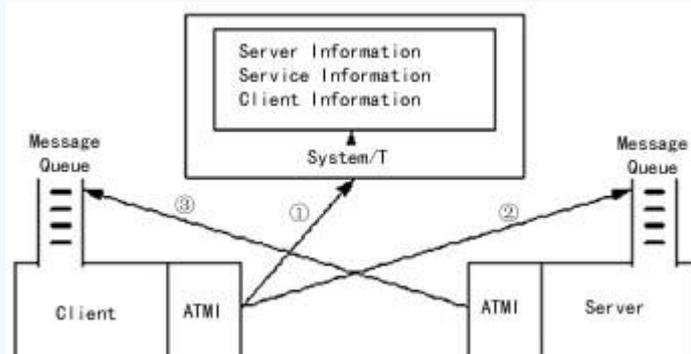


图 2

Client 向 System/T 发出查询请求，以找到 Server 消息队列的地址；
Client 根据找到的入口地址将请求发送到 Server 的消息队列中；
Server 处理请求，并将结果返回给 Client 的消息队列。

1、server 端 ATMI (Application-to-Transaction Monitor Interface)

在 server 端 tuxedo 本身提供了一个标准的 main() 函数，它负责完成一些必要的工作。server 端编程只需要编写 service 处理函数，进行 service 的请求处理和回应处理。所以，在 server 端不需要调用 tpinit() 和 tpterm()。

(1)、TPSVCINFO

每个 service 函数都有统一的形式：

```
void tpservice (TPSVCINFO *svcinfo);
```

只有一个参数，该参数是指向 TPSVCINFO 结构的指针 (atmi.h)。该结构定义如下：

```
struct tpsvcinfo {  
    char name[32]; /*service 名 (最大 15 个字符) */  
    long flags; /* client 调用时指定的 flags */  
    char *data; /* 接收的数据地址 */  
    long len; /* 数据长度 */  
    int cd; /* 会话方式下的连接描述符 */  
    long appkey; /* 应用认证的 key */  
    CLIENTID cltid; /* client ID */  
};
```

(2)、buffer 管理

在 service 函数里，一样可以调用 `tpalloc()`、`tpfree()`、`tprealloc()` 处理自定义的 buffer。

通过 `TPSVCINFO` 参数传递的 buffer 是使用 `tpalloc()` 分配的，所以可以对它使用 `tprealloc()`。

要注意的是在 service 函数里自己调用 `tpalloc()` 分配的空间在退出要释放，除非该空间作为 `tpreturn()` 或 `tpforward()` 的参数。如果分配的空间不释放，最终会耗尽该 server 的内存资源。

对于 `TPSVCINFO` 传递的 buffer 不用手动释放。

使用举例：

```
void
BAL (TPSVCINFO* input)
{
    char* f, f1, f2;
    f=input->data;
    f1=tpalloc ("STRING", NULL, 80);
    f2=tpalloc ("STRING", NULL, 120);
    . . .
    tpfree ((char *) f2);
    tpreturn (TPSUCCESS, 0, f1, 0, 0);
}
```

可以使用 `tptypes()` 查看 buffer 的类型。如：

```
void ABAL(TPSVCINFO *transb)
{
    char type[20], subtype[20];
    long len;
    len = tptypes(transb->data, type, subtype);
```

```
if (len == 0) {
    /*error*/
    userlog( "NULL message sent...\n" );
    ...
}
if (strcmp(type, "FML" ) == 0) {
    /* convert FML to aud VIEW; */
} else if (strcmp(type, "VIEW" ) == 0) {
    if (strcmp(subtype, "aud" ) != 0) {
        /*error*/
        userlog( "Wrong VIEW subtype..." );
        ...
    }
} else {
    /*error*/
    userlog( "Invalid buffer type ..." );
    ...
}
}
```

(3)、server 的 client 角色

tuxedo 的机制允许一个 server 程序作为 client，调用 `tpcall()` 去请求别的 service 服务。这样做可以避免代码的冗余，但效率上会有牺牲。

这样做时有一些情况要注意：

如果一个 server 要把负责返回接收到的 buffer，则不要使用这个 buffer 作为参数来请求别的 service 服务。可以分配辅助性的 buffer 来处理，这样做可以避免在 `tpcall()` 过程中改变了输入 buffer 的内容和类型。

一个 server 不能请求一个只被它本身发布的 service。这样做会导致死锁。例外是请求时指定 TPNOREPLY 标志。

(4) 、tpadvertise

service 可以在启动时发布,也可使用 tadmin 或使用 tpadvertise 动态发布。

```
int tpadvertise (char *svcname, void (*func) (TPSVCINFO *));
```

参数说明:

svcname: 要发布的 service 名;

func: 该 service 对应的处理函数指针;

如果该 service 用 func 已经发布,则函数立即成功返回。如果调用 server 是 MSSQL 集的一员,则该 MSSQL 中的所有 server 都发布这个 service。失败时返回-1。

出错原因:

该 service 已经用别的函数发布了;

超过了最大允许发布的 service 数量 (MAXSERVICES) ;

参数错误 (有为 NULL 的) ;

协议错;

操作系统错等。

(5)、tpunadvertise

一个 server 取消发布一个它已经发布的 service。

```
int tpunadvertise (char *svcname);
```

参数说明:

svcname: 操作的 service 名;

如果调用 server 是 MSSQ 集的一员, 则该 MQSQ 中的所有 server 都取消发布这个 service。失败时返回-1。

失败原因:

service 没有发布;

参数错误;

协议错等。

(6)、tpreturn

普通的 C 函数返回时使用 return 语句。但在 tuxedo 程序里, 不能使用 return, 必须使用 tpreturn() 终止当前处理并发送回应给请求端; 或者使用 tpforward() 把请求传递给别的 service 处理。

```
void tpreturn(int rval, int rcode, char *data, long len, long flags);
```

参数说明:

rval: 返回值, 决定该 service 请求是否成功。如三个可选值:

TPSUCCESS: 成功。tpcall 和 tpgetrply 将返回一个非负值; 如果是会话 service, 则产生 TPEV_SVCSUCC 事件;

TPFAIL: 失败。tpcall 和 tpgetrply 将返回-1; tperrno 设置为 TPESVCFAIL, 如果是会话 service, 则产生 TPEV_SVCFAIL 事件;

TPEXIT: 除了 TPFAIL 的功能外, 该 server 随后将终止, 如果配置了重新启动, 则该 server 将重新启动。

rcode: 应用定义的返回码。service 将向 client 返回该数字, tuxedo 本身不对其做任何解释, 接收端通过全局变量 tpurcode 得到该值, 在成功或失败时该值都会返回;

data: 返回给 client 端的数据 buffer;

len: 返回数据的长度 (只 CARRAY 类型有用);

flags: 标志, 当前没用。

(7) 、tpforward

调用 tpforward 的 server 不向 client 返回数据, 而是把更新过的 buffer 传递给另一个 service 做更多的处理, 由它处理返回 client 数据等后续的工作。本身则返回到标准的 main 流程中。

```
void tpforward(char *service, char *data, long len, long flags);
```

参数说明:

service: 后续的 service 的名称;

data: 指向传递的 buffer 的指针;

len: buffer 的长度 (只 CARRAY 有用);

flags: 当前没有使用;

该函数不能用在会话 service 中。

(8)、tpsvrinit/tpsvrdone

在一个 server 的实例启动时会调用 `tpsvrinit()`，在结束时会调用 `tpsvrdown()`。如果应用没有定义这两个函数，则使用 tuxedo 提供的缺省函数。

```
int tpsvrinit(int argc, char **argv);
```

参数说明：形式类似与 main 函数的参数，函数里可以使用 `getopt` 和全局变量 `optind` 进行处理，配置文件中的 `CLOPT` 命令行选项也传递到该函数。

函数成功返回 0，失败返回-1。

```
void tpsvrdone();
```

2、Tuxedo Client 端 ATMI

(1)、tpchkauth

```
int tpchkauth();
```

在调用 `tpinit()` 之前检查是否需要认证和认证的级别。

返回值：

- * `TPNOAUTH`：不需要认证；
- * `TPSYSAUTH`：系统认证，需要密码；
- * `TPAPPAUTH`：应用认证，需要密码和特殊应用数据；

当返回值为 `TPSYSAUTH` 和 `TPAPPAUTH` 时，我们必须使用 `tpalloc()` 分配一个 `TPINIT` 结构，在该结构中填入认证数据，然后用该结构作为参数调用 `tpinit()`。

失败原因主要有：

- * 协议错；
- * 操作系统错；

* tuxedo 底层错。

(2) 、tpinit

在使用 tuxedo 其他服务之前，必须调用 tpinit 加入到应用中。

```
int tpinit(TPINIT *tpinfo);
```

参数说明：

tpinfo: 指向 TPINIT 类型的指针。

TPINIT 类型在 atmi.h 中有定义，如以下几个域：

```
char username [32]; (32 characters significant)
char cltname [32]; (32 characters significant)
char passwd [32]; (8 characters significant)
char grpname [32]; (32 characters significant)
long flags;
long datalen;
long data;
```

username: 用户名或 login 名；

cltname: 应用定义；

passwd: 应用密码；

grpname: 在事务中使用，必须在配置文件定义的组列表中；

flags: 定义请求/通知类型和系统存取方法，其中 TPU_SIG、TPU_DIP 和 TPU_IGN 不能同时指定；TPSA_FASTPATH 和 TPSA_PROTECTED 不能同时指定。有如下的值：

- * TPU_SIG: 选择信号通知；
- * TPU_DIP: 选择 dip-in 通知；
- * TPU_IGN: 忽略通知；
- * TPSA_FASTPATH: 选择 fastpath 方式系统存取；
- * TPSA_PROTECTED: 选择 protected 方式系统存取；

datalen: 应用特殊数据的长度；

data: 应用特殊数据；

域 `flags` 的值覆盖系统的缺省定义,前提是在配置文件中没有指定 `NO_OVERRIDE`。
如果参数使用 `(TPINIT*)NULL`, 则 `client` 使用系统缺省的通知设置和系统存取设置, 若需要认证, 则出错返回 `TPEPERM`。

`tpinit()` 调用失败返回 `-1`, 失败原因有:

- * 参数错;
- * 没有空间在 `BB`;
- * 没有权限;
- * 协议错;
- * 操作系统错;
- * `tuxedo` 底层错。

示例:

```
TPINIT *tpinfo;
char password[9];
/* prompt user for password */
if ((tpinfo = (TPINIT *)tpalloc(“TPINIT”, NULL,
                                TPINITNEED(0))) == NULL) {
    (void)userlog(“unable to allocate TPINIT buffer”);
    exit(1);
}
(void)strcpy(tpinfo->passwd, password);
(void)strcpy(tpinfo->usrname, “Smith”);
(void)strcpy(tpinfo->cltname, “Teller”);
tpinfo->flags = (TPU_DIP|TPSA_PROTECTED);
if (tpinit(tpinfo) == -1) {
    (void)userlog(“failed to join application”);
    tpfree((char*)tpinfo);
    exit(1);
}
```

(3)、tperm

使用 tuxedo 服务完毕，调用 tpterm() 离开应用。

```
int tpterm();
```

函数出错返回-1。

错误原因有：

- * 协议错；
- * 操作系统错；
- * tuxedo 底层错。

(4)、tpacall

发送异步请求。

```
int tpacall(char *service, char *bufptr, long length,  
long flags);
```

参数说明：

service: 请求的 service 名(最大 15 个字符，以 null 结尾)；

bufptr: 请求发送的数据；

length: 发送数据长度(只有 CARRAY 类型用，其他设为 0)；

flags: 发送模式，有如下的值：

- * TPNOTRAN: 该次调用不能在一个事务里；
- * TPNOREPLY: 不需要回应(reply)；
- * TPNOBLOCK: 非阻塞；
- * TPNOTIME: 不超时，一直等待；
- * TPSIGRSTRT: 被信号中断的系统调用重启。

成功返回一个非负的描述符，该描述符可用于后续的 tpgetrply 调用，出错返回-1。

错误原因有：

- * 参数错；

- * 当前太多的 tpacall 处理存在，上限是 50；
- * 事务错；
- * 超时(time-out)；

(4)、tpgetrply

接收异步回应数据。

```
int tpgetrply(int *handle, char **bufpp, long *length,  
long flags);
```

参数说明：

handle: tpacall 返回的描述符；

bufpp: 接收 buffer 的地址的地址，原 buffer 会自动调整；

length: 接收的 buffer 的长度的地址；

flags: 接收选项。有如下值：

- * TPNOBLOCK: 非阻塞；
- * TPNOTIME: 不超时，一直等待；
- * TPSIGRSTRT: 被信号中断的系统调用重启；
- * TPGETANY: 接收任何回应；
- * TPNOCHANGE: 要求接收的回应与发送数据相同。

成功返回 0，失败返回-1。

出错原因:

- * 参数错;
- * 错误的接收 buffer 类型;
- * 超时;
- * 其他错误;

(6) .tpcancel

取消由 tpacall 发送的请求的响应, 在没有事务未完时。不能取消一个已经处理的请求。

```
int tpcancel(int handle);
```

参数说明:

handle: tpacall 返回的描述符;

出错返回-1。错误原因有:

- * 错误的描述符;
- * 当前在事务模式;
- * 其他错误;

(7) 、tpcall

同步发送请求并接收回应数据。

```
int tpcall(char *service, char *sbufp, long slength, \
           char **rbufpp, long *rlength, long flags);
```

参数说明:

service: 请求的 service 名;

bufp: 发送 buffer 的地址;

slength: 发送数据长度(只 CARRAY 使用, 其他为 0);

rbufpp: 响应 buffer 的地址的地址, 可以与发送 buffer 为同一块区域;

rlength: 响应 buffer 的长度的地址(不能为 NULL);

flags: 标志。有如下值(含义见 tpacall 和 tpgetrply):

- * TPNOTRAN
- * TPNOCHANGE
- * TPNOBLOCK
- * TPNOTIME
- * TPSIGRSTRT

返回-1 表示出错, 其他返回值都表示成功。

错误原因与 tpacall 和 tpgetrply 相同, 除了描述符错。

(8) 、tpgprio

获得上一次请求或接收的消息的优先级。

```
int tpgprio();
```

成功返回的范围是 1-100, 值越大优先级越高。失败返回-1。

使用举例:

```
struct {  
    int hdl; /* handle*/  
    int pr; /* priority*/  
} pa[SIZE];  
for (i=0; i < requests; i++) {  
    /* Determine service and data for request */  
    pa.hdl = tpacall(Svc, buf, len, flags);  
    /* Save priority used to send request */  
    pa.pr = tpgprio();  
}  
/* Use qsort(3) routine to sort handles in priority order */
```

```
qsort((char*) pa, requests, sizeof(pa[0]), cmpfcn);  
for (i=0; i< requests; i++) {  
    tpgetrply(&pa.hdl, &rbufp, &rln, rflags);  
}
```

(9)、tpsprio

设置下一个要发送的消息的优先级。

```
int tpsprio (int prio, long flags);
```

参数说明:

prio: 要设置的优先级;

flags: 标志。有如下值:

- * 0: 使用相对优先级, 值改为(default+prio);
- * TPABSOLUTE: 绝对优先级, 值改为 prio;

优先级的范围是 1-100, 超过次限制的值被改为相应的最大(小)值。

失败返回-1。错误原因有 TPEINVAL、TPEPROTO、TPESYSTEM、和 TPEOS。

System/T 是 Tuxedo 系统的核心, 它实现了 Tuxedo 的所有功能和特征, 如 C/S 数据流管理、服务请求的负载均衡、全局事务管理以保证交易的完整性、同步/异步服务请求、两阶段提交以确保消息的发送等。System/T 提供了一个类似公告栏的服务, 用以发布 C/S 计算机环境中所有服务器、服务和客户机的信息, 供其它分布式计算的参与者使用。下面笔者将通过一个大写字母转换的简单例子, 讲述 Tuxedo 应用程序工作的基本原理和开发方法。

(二)、tuxedo 的示例

Simpapp 是 Tuxedo 系统自带的一个例子, 它由服务器和客户端程序两部分组成。服务器 simpserv 实现了一个 TOUPPER 服务, 它从客户程序接收一个字符串, 将它转换成大写后, 传回客户端。整个工作流程可以用图 2 表示。

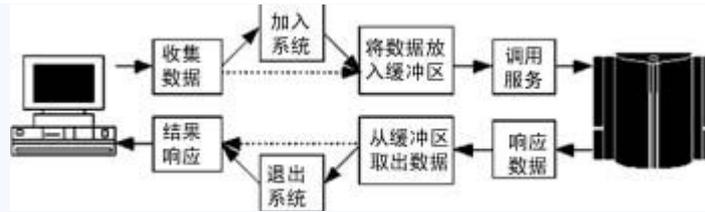


图 3

客户机首先收集要转换成大写的字符串，然后连接到 System/T 并将字符串放入缓冲区，接着调用服务器上的 TOUPPER 服务，最后从服务器响应缓冲区中取出数据并退出 System/T。

1. Simpsapp 的服务程序

下面是服务程序 `simpsserv.c` 的源代码：

```
#include <stdio.h>
#include "atmi.h"
/ Tuxedo ATMI 函数库的头文件 /
TOUPPER(TPSVCINFO rqst) {
int i;
for(i=0;i<rqst->len-1;i++)
rqst->data[i]=toupper(rqst->data[i]); / 将 rqst->data 缓冲区内容转
换成大写 /
tpreturn(TPSUCCESS, 0, rqst->data, 0L, 0); / 将 rqst->data 作为响应缓冲
区返回 /
}
```

可能你已经注意到了，该程序没有提供 `main` 方法。事实上，Tuxedo 不要求程序员编写 `main` 方法，以便让他们把精力集中在业务逻辑的编写上，在编译时，Tuxedo 系统会为它自动创建。

TPSVCINFO 是 Tuxedo 记录服务调用信息的一个结构体，data 域是保存请求数据的缓冲区，len 域记录了 data 域的长度。

2. Simpapp 的客户程序

客户程序 Simpcl.c 是服务调用的发起者，它从命令行接收参数，通过 tpinit() 调用连接到 System/T，通过 tmalloc() 调用分配一个字符串类型的缓冲区，通过 tpcall() 调用请求 TOUPPER 服务，最后通过 tpterm() 调用退出 System/T，下面是它的源代码：

```
#include <stdio.h>
#include "atmi.h"
int main(int argc, char argv[]) {
    char buf;
    long len;
    if(argc != 2) {
        (void) fprintf(stderr, "Usage: simpcl string\n");
        exit(1);
    }
    if (tpinit((TPINIT ) NULL) == -1) {
        (void) fprintf(stderr, "Tpinit failed\n");
        exit(1);
    }
    len = strlen(argv[1]);
    if((buf = (char ) tmalloc("STRING", NULL, len+1)) == NULL) {
        (void) fprintf(stderr, "Error allocating send buffer\n");
        tpterm();
        exit(1);
    }
}
```

```
(void) strcpy(buf, argv[1]);
if(tpcall("TOUPPER", buf, 0, &&buf, &&len, 0)==-1){
(void) fprintf(stderr, "Can't send request to service TOUPPER\n");
tpfree(buf);
tpterm();
exit(3);
}

(void) fprintf(stdout, "Returned string is: %s\n", rcvbuf);
tpfree(buf);
tpterm();
return(0);
}
```

3. Simpapp 的配置文件

除了客户和服务程序以外，Tuxedo 还需要一个配置文件来对应用进行描述。配置文件由多个段组成，每个段定义由一个星号开始。下面是 NT 平台下 simpapp 的配置文件 ubbsimple 的内容，其中带下划线的部分需要根据机器的资源配置作适当修改。

```
RESOURCES
IPCKEY 123456
MASTER NODE1
MODEL SHM
MACHINES
JQ LMID=NODE1
TUXDIR="XX:\ProgramFiles\BEA Systems\Tuxedo"
APPDIR="XX:\simpapp"
TUXCONFIG="XX:\simpapp\tuxconfig"
```

```
GROUPS
```

```
GROUP1 LMID=NODE1 GRPNO=1
```

```
SERVERS
```

```
simpserv SRVGRP=GROUP1 SRVID=1 CLOPT="-A"
```

```
SERVICES
```

```
TOUPPER
```

TUXDIR 指的是 Tuxedo 的安装路径, APPDIR 指的是 simpapp 应用程序所在的目录, TUXCONFIG 指的是 simpapp 的二进制配置文件, 一般为%APPDIR%\tuxconfig。

4. Simpapp 的编译和运行

由于 Buildclient 和 Buildserver 没有编译能力, 要编译 Tuxedo 应用程序时, 还必须安装第三方的 C 语言编译器, 在 NT 平台下推荐使用 VC。步骤如下:

(1)、设置环境变量

```
SET PATH=%PATH%; G:\Program Files\BEA Systems\Tuxedo\Bin
```

```
SET TUXDIR=G:\Program Files\BEA Systems\Tuxedo
```

```
TUXCONFIG=G:\simpapp\tuxconfig
```

(2)、生成二进制配置文件

```
tmloadcf -y ubbsimple
```

(3)、编译客户程序

```
buildclient -o simpcl.exe -f simpcl.c
```

(4)、编译服务程序

```
buildserver -o simpserv.exe -f simpserv.c -s TOUPPER
```

(5)、启动服务程序

```
tmboot -y
```

(6)、运行客户程序

```
simpcl "hello, JQ"
```

(7)、关闭应用程序

```
tmshutdown -y
```

5. 管理 Simpapp

通过 `tmadmin` 命令可以方便地管理服务程序。

(三)、Tuxedo 核心函数 (Service) 的测试

AMTI 的作用就是提供接口调用实现业务逻辑的 Service。所以真正的核心代码就是这些实现业务逻辑的 Service。这些 service 基本上都是 C/C++ 的。可以说银行核心业务系统的测试都是围绕着这些 Service 来进行的。这些 Service 一般都是部署在 Tuxedo 的服务器上, 这些服务器绝大多数是 Unix 系统的小型机, 甚至大型机。

就目前常用的测试方式, Tuxedo 核心函数的测试有下面几种方式:

1、自动化测试工具的使用

(1)、Tuxedo 协议

网上有很多类似的 Loadrunner 测试部署参考, Rational 也有类似的使用 Rational

(http://www.ibm.com/developerworks/rational/library/05/1025_jong-straathof/index.html;

<http://www.oracle.com/technology/pub/articles/dev2arch/2006/06/performance-tuxedo.html>)。

应该说这是目前最常推荐的用法，这样对测试人员要求相对低，对开发人员的依赖程度比较低。但是有些问题相对有些限制。目前的 Tuxedo 基本上都是部署在 Unix 和 Linux 服务器上，C/S 结构的特点导致我们录制 Service 对应的 shell（对应于 windows 下的 exe 文件）时需要部署 Loadrunner 的端口代理（因为 loadrunner 没有基于 Unix 的版本），例如当我们要录制一个部署于 AIX 系统的 Tuxedo 的 Service 调用。（流量转发允许从本地端口转发到目标此选项将来自特定端口的所有流量转发到另一个服务器）例如，如果您在名为 host1 的 UNIX 客户端上工作，该客户端与服务器 server1 通过端口 8080 通信，您可以为 server1、端口 8080 创建一个“端口映射”项。在“服务器项”对话框的“流量转发”部分中，通过选中“允许从本地端口转发到目标服务器”复选框启用流量转发。在本例中，将想要从其转发流量的端口指定为 8080。然后，将客户端 host1 连接到运行 VuGen 的计算机，而不是连接到 server1。VuGen 从客户机接收通信，并通过本地端口 8080 将其转发到服务器。由于流量流经 VuGen，因此 VuGen 可以分析流量并生成相应的代码。

(2)、封装为 DLL，提供给工具调用。

Loadrunner 的 DLL 调用测试我们已经验证。其特点是需要开发人员的参与，协助把核心函数封装为 DLL。

2、复制进程实现对客户端进程的复制

（相当于直接在 unix 下编写 Shell 脚本实现，不用第三方的测试工具，一般开发人员的参与编写进程的代码。优点就是也是和核心函数的编程语言一致，可以直接嵌套源代码。）对 TUXEDO 服务器而言，每一个客户端就是一个和它通信的进程，所以需要多少客户端简单的说就是开多少调用服务的进程，这个在 Unix 操作系统下是很容易实现的。（利用 Fork 函数可以实现进程复制）。关键是这些进程的代码编写得靠开发人员提供和协助调试，另一个缺点是对测试的可控性比较差，并发控制，事务交易统计等。

```
for ( i = 0; i < P_NUM; i++ )           //P_num 为开辟的进程数  
量  
{  
    if ( (pid = fork()) == 0 ) break;    /*  
    if ( pid < 0 ) exit(0);  
}  
    if ( pid == 0 )                      //子  
进程代码  
{  
    child_process();  
}  
    if  
( pid )                                //  
父进程代码  
{  
    ...  
}
```

下图可以表示为实现的过程和数据流

