OSF：一种支持SaaS应用的构件框架

宋 杰 侯泓颖 朱志良
（东北大学软件学院 沈阳 110189）

摘 要 离线SaaS应用可以工作在时断时续的网络连接环境下，是一种环境可感知的智能SaaS软件。目前支持离线访问Web应用的研究成果较少，且其没有考虑到SaaS的特性，更没有形成构件化框架。为了解决上述问题，提出了离线SaaS框架OSF(Offline SaaS Framework)，并给出了支持离线访问的面向操作的构件框架的结构和运行机制。通过案例描述验证了该构件框架的功能和性能。理论和实验表明，离线SaaS应用框架极大地改善了用户体验，保证了SaaS服务的可用性，扩大了SaaS的应用范围，同时还提高了系统开发效率。

关键词 SaaS，离线应用，构件，框架
中图法分类号 TP301.41 **文献标识码** A

OSF：A Component-based Framework Supporting Offline SaaS Applications

SONG Jie HOU Hong-ying ZHU Zhi-liang
(Department of Software College, Northeastern University, Shenyang 110189, China)

Abstract Offline SaaS application is environment-appreciable smart software which can work with discontinuity connection. Currently there are a few research works about the offline Web application. They can't meet the requirements of SaaS, and can't take componentized framework into account. To solve the problems above, OSF (Offline SaaS Framework) was proposed. Firstly, the structures and mechanisms of operation-oriented component-based offline SaaS framework were proposed. And then a case study was introduced to verify the proposed framework. Theories and practices show that offline SaaS application framework promotes the users' experience, enforces the usability of SaaS service, and enhances the applied area of SaaS applications.

Keywords SaaS, Offline application, Component, Framework

1 引言

SaaS是云计算领域中备受瞩目的软件交付模式，其交付能力是基于Web的。因此我们认为，首先，SaaS是一种基于Web的应用模式，称为SaaS应用；其次，SaaS是一种服务。服务的可用性是衡量服务质量的重要因素，SaaS服务的可用性首先要基于良好的网络连接。在很多环境下，用户必须频繁地在有网络连接和无网络连接的工作环境中切换，并一直运行应用系统。比如销售人员在客户所在地运行销售系统时，必须在无连接的环境下运行软件。用户还可能因为网络拥挤或者时断时续的低质量网络环境而被迫离线。在上述情况下，SaaS应用应该在连接存在的时候充分利用服务器资源，在不影响运行效率的同时在后台同步客户端和服务器状态；当连接断开后，SaaS客户端应该缓存状态并等待下一次连接。

基于Web的应用系统从静态封闭逐步走向动态开放。为了适应这样一种发展趋势，Web应用系统在软件构件技术支持下开始呈现出一种柔性、多目标、连续反应式的新系统形态，它可感知外部网络环境的动态变化，并随着网络环境的变化按照功能指标、性能指标和可靠性指标等进行动态的演化，以使系统具有高可用性和良好的用户体验。本文研究的支持离线操作的SaaS应用恰属于这一领域。目前支持离线访问的应用程序主要分为两种，一种是通用的浏览器插件或浏览器本身，另外一种则是基于客户端-服务器方式的C/S应用程序。前者提供的离线功能有限，后者则非基于Web的应用程序。现有研究多采用面向数据的实现方式，依赖浏览器或是客户端本地存储来实现，这种方法受数据同步的限制，难以支持复杂应用逻辑，且没有考虑到SaaS的特性，也没有形成构件化框架。

在桌面应用程序领域，Poznan超级计算和网络中心[1]构建的Offline Business Objects (OBO) 框架和Phu-Nhan Nguyen[2]等提出的小型业务模型(Small Business Model)实现了在偶尔连接网络环境下运行的功能。Web应用比桌面应用程序更适于SaaS交付模式和云计算环境，更有应用背景。在Web应用程序领域，文献[3]提出离线SaaS应用的需求和挑战。文献[4]较早地提出了离线应用的概念，并提供了

到稿日期：2012-02-14 返修日期：2012-03-15 本文受国家自然科学基金(61173028)，中央高校基本科研业务费专项资金(N110417002)，辽宁省自然科学基金(200102059)资助。

宋 杰(1980－)，男，博士，副教授，主要研究方向为云计算和高能效计算，E-mail：songjie@mail.neu.edu.cn；侯泓颖(1988－)，女，硕士生，主要研究方向为云计算和高能效计算；朱志良(1963－)，男，教授，博士生导师，主要研究方向为Web服务和复杂网络。

一个基于 Java Applet 的离线客户端，但其没有考虑数据的冲突问题。文献[5]提出一个离线 RSS 浏览器，其可以在在线时抓取 RSS 和邮件，供离线时访问。文献[6]使用 HTML5 在手机上实现了支持离线 Web 访问的 e-learning 客户端。HTML5 可以用来实现离线 SaaS 应用，但 HTML5 仅提供一种支持离线的语言标准，还需要浏览器的支持。此外，Google 公司提供的 Gear 浏览器的插件[7]、Edgar Gonçalves[8]实现的基于 Web 的可离线执行的工作流系统也都与本研究相关。

总体上，目前研究未能良好地支持 SaaS 应用的离线功能，该研究问题存在以下挑战：

• 探索实现离线 SaaS 应用的方式和方法，总结和归纳其体系结构和可行性；

• 为支持离线操作，定义和解决对请求的处理、缓存的实现、操作同步、依赖和冲突检测问题；

• 离线应用的实现方法如何充分针对 SaaS 应用的多租户特性实现客户端租户和用户数据的隔离，以方便离线状态的服务可配置和计费；

• 为方便二次开发，离线功能应封装良好，以构件的方式提供给 SaaS 应用开发人员。

因此，本研究提出了离线 SaaS 框架 OSF(Offline SaaS Framework)。它可感知外部网络连接的动态变化，并随着网络连接的变化改变自己的运行方式，在网络拥挤或是完全断开的情况下正常运行，从而不间断地为用户提供服务。OSF 由若干构件组成，基于该框架，二次开发人员可以很容易地实现支持离线运行的 SaaS 应用程序。其创新在于：首先它是一个完整的框架和解决方案，而不仅是部分关键算法；其次它是基于 Web 的 SaaS 应用，而不是桌面应用或是浏览器；再次它是一种基于构件的框架，在实现离线 SaaS 应用框架的同时提供了更高的抽象和更好的通用性。

2 构件框架结构和原理

SaaS 应用是一种基于 Web 的应用，我们设计了 3 种不同结构的离线 SaaS 应用框架：面向数据的构件框架、面向操作的构件框架、面向服务的构件框架。面向数据的构件框架的核心思想是客户端利用本地数据库来管理对本地保存的数据所作的更改，然后使用合并-复制机制将这些更改传回服务器端数据库。其框架结构简单，编程工作较少，但是需要在客户端安装一个本地数据库，以便和服务器进行复制。面向服务的构件框架中客户端是以服务的方式与服务提供者进行交互的，但是由于 SaaS 服务的接口、协议尚未规范化，面向服务的构件框架并不能很好地适用于 SaaS。面向操作的构件框架基于操作的不同实例之间的版本比较和同步，其机制更灵活，更具有研究性。因此，本文详细介绍面向操作的构件框架。以下是关于离线 SaaS 应用的基本概念。

定义 1(间断性连接，Discontinuous Connection) 设客户端和服务器存在连接的状态称为 α 态，不存在连接的状态称为 β 态。设函数 $fc(t)$ 表示取 t 时刻的连接状态，显然 $fc(t)=\alpha$ 或 $fc(t)=\beta$。若从任意时间段 T 中选择 n 个时间点，有：$\bigcup_{i=1}^{n} fc(t_i)=\{a,b\}$，则认为 T 时间段是间断性连接。

定义 2(离线 SaaS 应用，Offline SaaS Applications) 离线 SaaS 应用是一种可以在间断性连接环境下正常工作的 SaaS 应用，除正常的 SaaS 应用功能外，还包括以下几个功能：

• $fc(t)$ 函数的实现。

• 当 $fc(t)=\alpha$，客户端与服务器自动同步数据和状态。

• 当 $fc(t)=\beta$，利用缓存不间断地提供应用服务。

图 1 展示了面向操作的构件框架的总体结构。面向操作的构件框架中引入了操作的概念，一个操作是一次业务逻辑的抽象，一个操作有多个实例，每个实例包含着该实例执行时用到的数据的集合。

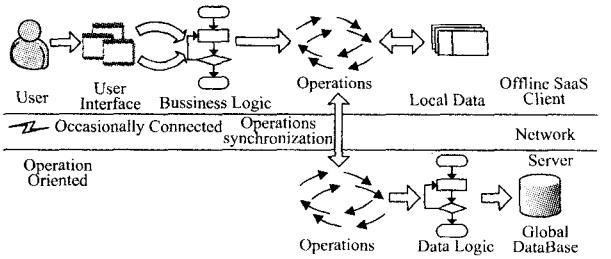

图 1 面向操作的框架的总体结构

定义 3(操作，Operation) 一个操作 op 是特定业务逻辑的抽象。操作应该是原子的，设 OP 为所有操作的操作集，$\forall a,b\in OP$，如果 a 和 b 分别可以分解成 n_i 和 n_j 个子操作，当 $a=\{a_i\mid 0<i<n_i\}$，$b=\{b_j\mid 0<j<n_j\}$，且 $a\cap b=\emptyset$，则认为 a 和 b 都是原子操作，进而 OP 集中所有操作都是原子的。

定义 4(实例，Instance) 设 $\overline{op}$ 是操作 op 的实例集，则 $\overline{op}=\{\overline{op_i}\mid 0<i<n, n \text{ 是 } op \text{ 的实例的个数}\}$，$\overline{op_i}$ 是一个三元组 $\langle Logic, Data, Version\rangle$，其中 $Logic$ 表示该操作封装的业务逻辑，$Data$ 表示该操作封装的数据，$Version$ 用来标识当前操作的版本号。

本文把存在于服务器端的实例称为 α 实例，存在于客户端的实例称为 β 实例。有且仅有一个 α 实例稳定存在于服务端(合并-复制时可能会同时存在多个，但多余的很快被销毁)，客户端若存在，也只能存在一个 β 实例。客户端请求操作时，在 α 态可以连接到服务器，因此执行 α 实例，在 β 态由于无法连接，只能执行客户端 β 实例。

定义 5(版本，Version) 操作实例的版本代表了实例的状态。若 $\forall op\in OP, a, b$ 是 op 的两个实例，$a=b\Leftrightarrow a.version=b.version$。同一操作所有实例都有相同的逻辑，但是只要版本不同，其状态就不相同，包含的数据也不同，因此 $Version$ 是指示 $Data$ 以及整个实例状态的标志。

面向操作构件框架的特征有：①操作的实例是合并、冲突检测和处理的基本单元，这个粒度非常接近面向对象的思想，是现实逻辑的很好的抽象；②一个操作的实例既存在于服务器端又存在于客户端，但是它们的版本不同；③业务逻辑以及其数据被很好地封装在客户端，充分利用了本地硬件和资源。该方式也需要一个客户端缓存，它既可以是数据库也可以是文件系统或内存，因此这个方式理论上适用于任何客户端设备，如计算机、智能手机或是 PDA。所有与离线相关的变更管理或冲突检测机制都依赖于实例的版本比较和手工编码，与依赖于 DBMS 的面向数据构件框架相比较，手工编码的方式更加灵活。

3 构件框架的实现

OSF 的设计遵循了定义 2 中对离线 SaaS 应用程序的定义。尽管本文采用面向操作的构件框架来实现 OSF，但其一些设计方法对其它构件框架同样有参考价值。

3.1 操作缓存构件

OSF 确保在 β 态仍可以继续执行操作，因此所有的操作都存在若干 β 实例，这些 β 实例存储在各个客户端的缓存中。操作缓存构件就是管理这些 β 实例的构件。

定义 6（无状态操作，Stateless Operation） $\forall op \in OP$，$\forall a \in \overline{op}$，在 a 的生命周期时间序列上任取 $t1, t2$ 两个时间点，若 $a^{t1}.Version = a^{t2}.Version$，则认为 a 是一个无状态操作实例，进而 op 是一个无状态操作。

无状态操作并非没有状态，而是其状态不会被客户端改变。显然，无状态操作的实例能够储存在客户端。通过存储和使用本地无状态操作的实例，客户端与服务器的交互和数据传递明显减少。因此，缓存无状态操作实例不仅能够提供离线功能，而且能够提高应用程序的性能，充分利用客户端资源。无状态操作虽然不能被客户端更改，却可能在服务器端更改，因此无状态操作也存在同步需求。

定义 7（有状态操作，Stateful Operations） $\forall op \in OP$，$\forall a \in \overline{op}$，在 a 的生命周期时间序列上任取 $t1, t2$ 两个时间点，若存在 $a^{t1}.Version \neq a^{t2}.Version$，则认为 a 是一个有状态操作实例，进而 op 是一个有状态操作。

相对于无状态操作，有状态操作的状态可以在服务器端也可在客户端中更改。一般地，状态改变是由用户输入和执行而引起的直接或间接的结果。OSF 应该跟踪任何客户端有状态操作实例所作的更改，在这种情况下，无论是 β 实例还是其对应的 α 实例状态的改变都应该在某一时间点同步。

此外，对于 SaaS 应用，租户信息在用户间共享，任何操作都包含了租户信息。OSF 假定所有的租户信息都是无状态的，用户在 β 态时不可更新租户信息或是更换租户。这种假设是成立的，租户信息稳定，很少出现某用户属于多租户并频繁更换租户信息的情况；并且这种假设也是有必要的，否则所有操作都将是有状态的。

无论是无状态操作还是有状态操作，其实例都是存储在缓存中的，OSF 中的缓存分长期缓存（Long-term Cache）和短期缓存（Short-term Cache）。长期缓存是持久化的存储，其存储能力较大但执行效率不高，在系统运行时长期缓存用来存储较稳定的无状态操作的 β 实例。短期缓存是非持久化的存储，其存储能力有限但执行效率很高，短期缓存用来存储易变的有状态操作的 β 实例。当客户端关闭后短期缓存即会消失，因此需要将实例转换到长期缓存中。OSF 选择内存作为短期缓存的实现，选择文件系统（也可以是数据库系统）作为长期缓存的实现。

3.2 操作同步构件

操作同步构件是面向操作的构件框架中最重要的一个构件，用来处理 α 实例和 β 实例之间的同步。操作同步构件还要对冲突进行检测和处理。当冲突不可避免地发生时，操作同步构件就要保证这些冲突可以检测到，并且是可以解决的。

定义 8（操作同步） 操作同步是指同一操作的 α 实例和 β 实例之间统一状态的过程，操作同步分为 3 种方式：正向同步、逆向同步和合并同步。设 α 实例 $ins\alpha$ 和 β 实例 $ins\beta$ 同属于一个操作，其中：

- 正向同步是指 $ins\beta$ 替换 $ins\alpha$，即客户端在 β 态执行了某个操作，在 α 态下同步回服务器，记作 $ins\beta \Rightarrow ins\alpha$；
- 逆向同步是指 $ins\alpha$ 替换 $ins\beta$，即客户端在 α 态更新了某个陈旧的 β 实例，记作 $ins\beta \Leftarrow ins\alpha$；
- 合并同步是指 $ins\beta$ 和 $ins\alpha$ 合并成一个新的实例，即某客户端在 β 态执行了 $ins\beta$，同时其他客户端在 α 态正向同步了 $ins\alpha$，这时 $ins\beta$ 无法与 $ins\alpha$ 进行正向同步而需要合并两次变化，记作 $ins\beta \Leftrightarrow ins\alpha$。

不失一般性，$\forall op \in OP$，$\forall a \in \overline{op_\alpha}$，$\forall b \in \overline{op_\beta}$，即 a、b 分别是操作 op 的一个 α 实例和 β 实例。由定义 7 可知，$a.Version$、$b.Version$ 分别是实例 a、b 的版本，设 $a.Version$，$b.Version$ 的初始值都是 λ，同步公式如下：

$$\begin{cases} b \Rightarrow a(a.Version = \lambda, b.Version > a.Version) \\ b \Leftarrow a(b.Version = \lambda, b.Version < a.Version) \\ b \Leftrightarrow a(b.Version \neq a.Version \neq \lambda) \end{cases}$$

当 $b.Version \neq a.Version \neq \lambda$ 时，合并同步有可能产生 a、b 实例的矛盾性和不一致性。即产生合并冲突，在多用户的 SaaS 应用中这种合并同步发生的可能性比较高，因此如何协调冲突是操作同步构件的核心任务。

（1）管理过期实例

定义 9（过期实例，Expired Operation） 当一个 β 实例（α 实例）自身状态没有改变，而它对应的 α 实例（β 实例）发生状态变化时，则称该 β 实例（α 实例）为过期实例。沿用定义 8 的例子：

a is $Expired\ Operation \Leftrightarrow a.Version = \lambda, a.Version < b.Version$

b is $Expired\ Operation \Leftrightarrow b.Version = \lambda, b.Version < a.Version$

操作同步构件采用正向同步来更新服务器端的过期实例；采用逆向同步来更新客户端的过期实例。过期实例是相对的，β 实例相对 α 实例可以是过期实例，α 实例相对于某个客户端的 β 实例也可以是过期实例，因此过期可能同时存在于客户端和服务器端。

（2）操作分区和加锁

对操作进行分区、加锁和设定优先级的目的在于减少冲突，任何允许多用户共享的操作都有产生冲突的可能。如果采用操作分区的方式把同类操作分在一个集合中，客户端可以给某个区中的操作加锁来限制并发访问，这样有利于减少甚至在某种程度上避免冲突的发生。

定义 10（操作分区，Operation Partitioning） 操作分区是指按照操作的功能和相互关系把全部操作分成若干个子集合，设操作集 OP 可以被分解为 n 个区，即 OP_1 至 OP_n 集，这些区应该满足以下两个条件：

1）$OP_i \cap OP_j = \varnothing$ 或尽可能小（$0 < i, j < n$）。

2）同时访问 OP_i 的用户为 1 或尽可能少。

定义 11（操作加锁，Operation Locking） $\forall op \in OP$，在

时间序列上任取 $t1$，$t2$ 两个不相等的时间点，op^{t1} 表示 $t1$ 时刻的实例，op^{t2} 表示 $t2$ 时刻的实例。

op is *locked operation* $\Rightarrow op^{t1}=op^{t2}$

操作加锁是指使用互斥的排它锁来保证在一个时间点某操作只有一个实例在系统上运行。操作优先级表明多个用户同时操作一个操作时排队的顺序，即操作优先级高的用户先得到该操作的使用权。

操作分区、加锁和优先级控制的方式看起来简单，但很有效，而且实施起来是可行的。首先在 SaaS 应用中，所有的操作都可以按照其对应的租户分区，操作同步仅仅会在同一租户内进行；此外还可以按照业务逻辑进行划分，大多数的应用系统都有这种情形——不同的域控制着不同的操作集，而这些操作集是相互独立和分离的。对于所属操作区重叠的操作的访问可以通过操作优先级来解决。此外，只有在客户端进入 β 态时才有必要对操作进行加锁，对于间断性连接的 SaaS 应用实际产生冲突的可能性又降低了很多，用分区和加锁的方式来管理操作也降低了合并-复制的可能性，极大地简化了同步逻辑。

然而操作分区和加锁的限制性很强，难免会导致用户都在争抢一些共享程度很高的操作。对于这些操作来说，分区和加锁并不是一个很好的解决冲突的办法，所以不能完全依赖分区和加锁机制来解决冲突。尽管这样，OSF 仍然把这种方法作为一种重要的避免冲突的方法，因为它确实可以降低冲突产生的可能性。

(3) 协调冲突

操作分区和加锁的方式可以适当避免操作同步冲突，然而一旦冲突产生，必须有相应的协调办法。SaaS 应用中的冲突大部分都是业务逻辑相关的，所以当冲突确实发生时，OSF 应该尽可能多地提供有关冲突的详细信息以便进行自动处理，或者为用户或管理员提供足够的信息以便他能够协调冲突。操作同步构件能通过 3 种不同的策略来协调冲突。

策略 1(服务器端的自动协调)：操作同步构件首选在服务器端进行自动协调，需要为每一个有可能产生冲突的操作实现协调逻辑。也可以采用一些公用的、独立于业务逻辑的协调逻辑，如确保最新的更改总是优先策略，或是合并两个数据元素，或是采用更为复杂的协调逻辑。该策略能够使用户免于过多地涉及协调过程，明显提高了应用的可用性。

策略 2(客户端手动协调)：如果策略 1 不能协调冲突或协调过程过于复杂，操作同步构件就选用第 2 种策略：回发冲突到客户端，让操作者手工协调。要进行有效的客户端协调，服务器应该向客户端发送足够的数据，使客户端能够就如何解决冲突做出正确的决策。

策略 3(第三方的协调)：如果前面两个策略均不适合，操作同步构件就依靠第三方来协调冲突。在此情况下，客户端必须等待直至冲突解决为止(如果有可能，可以继续使用临时数据)。当该冲突解决后，将通知客户端，使其在收到协调结果后继续工作。

3.3 依赖处理构件

执行操作和同步操作时都需要考虑操作间的依赖性，这样才能保证流程的完整性。一般地，在顺序执行操作流程时如果其中一个操作节点失败了，与其相关的操作都需要进行回滚和补偿，这是传统事务系统中需要考虑的问题；同理，当顺序同步一系列操作时，也会出现因同步失败而产生的依赖处理。本文重点考虑同步操作时的依赖处理，OSF 通过依赖处理构件来解决同步操作时的依赖性问题。

定义 12(依赖性操作，Dependent Operation) 当一个操作实例同步成功与否取决于其它操作的同步结果时，称这种操作与其它操作有依赖性，即称为依赖性操作。

若存在一系列(n 个)的依赖性操作实例组成的一个同步队列，同步顺序为 $op_1, op_2, \cdots, op_n$，如果 $op_i(0<i<n)$ 操作同步失败了，op_{i+1} 至 op_n 操作的同步都要被忽略，则 op_{i+1} 至 op_n 操作的依赖性对 op_i 来说是正向依赖(Forward Dependency)；相对地，如果 op_i 操作同步失败了，op_1 至 op_{i-1} 操作的同步都要被撤销并回滚到原始状况，则 op_1 至 op_{i-1} 操作的依赖性对 op_i 来说是逆向依赖(Reverse Dependency)。

对于正向依赖的处理方法比较简单，即在同步的时候维护一个同步操作顺序的列表，当一个操作同步失败后，它的一系列的后续操作就应该被忽略，而逆向依赖会增加应用的复杂性。必须为同步的每一个操作备份原始的实例，以备回滚使用。事实上，在一次同步过程中正向依赖和逆向依赖是同时存在的，所以在进行操作同步时应该根据依赖特性，合理安排易失败操作的同步顺序。

此外，为了避免依赖处理，可以在应用构件开发时将相关的操作合并成同一个操作，把依赖关系封装到单一操作的内部。这种方法使依赖处理变得很简单，但是破坏了操作的原子性，影响了操作的移植性和重用性，增加了操作的复杂度。然而我们仍可以采用这种“外部处理”的方法解决复杂的逆向依赖。

3.4 租户隔离构件

租户隔离构件不同于其它构件，它用来在客户端本地保存用户和租户信息。当前用户和租户信息通常保存在服务器端会话中，在 β 态时，该会话将丢失，因此，前文介绍的所有 β 实例都需要单独保存用户和租户信息，以便再次进入 α 态时可以顺利同步。

在 SaaS 应用中，租户信息有以下功能：①关联服务器端(通常是数据库)的租户数据；②检查该租户授权使用的服务；③记录租户定制的界面和配置；④访问控制和计费。上述功能在 β 态时仍需正常使用，因此在客户端本地需要保存该客户端登录过的用户信息和租户信息。

用户和租户信息是无状态的，OSF 假设用户在 β 态时不能更新用户和租户信息，服务端数据恒为新版本，避免了冲突的发生。此外，缓存中的用户和租户信息存在一定的生命周期，若用户退出系统和切换用户，仅可以切换本地存在的用户，及近期在该客户端登录过的用户。租户隔离构件保证 β 态 SaaS 多租户特性的体现。

3.5 构件框架

作为一个构件框架，OSF 不仅实现了离线 SaaS 应用的功能，还为不同业务逻辑提供了扩展点。图 2 以构件图的方式说明了 OSF 的主要构件及其之间的关系，图中很多相对简单的构件文中不再详细介绍。

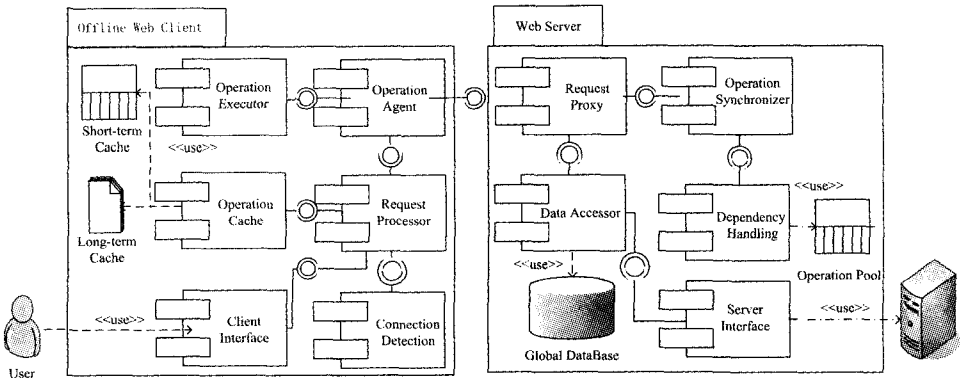

图 2　OSF 构件图

4　框架测试和案例分析

我们将著名的 Java Web 程序 JPetStore[9] 改造为 SaaS 应用，它增加了多租户特性，并且运用 OSF 实现了离线运行模式。本节对其进行了测试和性能分析。测试采用的客户端和服务器均为清华同方超翔 Z900 计算机，CPU 为 Inter Core i5-2300 2.80GHz，8GB 内存，1TB 硬盘。使用 Tomcat 6.0 作为 Web 容器，MySQL 作为数据库并包含了 20000 条测试数据。在服务器端按某条件拒绝特定请求，以模拟偶尔连接的网络环境。本节着重比较原始的 JPetStroe(Original)、支持离线访问和 SaaS 模式的 JPetStroe(OSF)的性能。衡量性能采用文献[10]提出的指标：

• 事务响应时间（TRT，Transaction Response Time）：客户端发送请求到得到响应之间经过的时间。

• 处理器时间(PT，Processor Time)：服务器端 CPU 在 Java 虚拟机、数据库等与 JPetStroe 相关进程中花费的时间与总工作时间的比例。

• 内存使用(MU，Memory Usage)：服务器端 Java 虚拟机、数据库等与 JPetStroe 相关进程耗费的内存总占有内存的比例。

测试采用“用户登录 JPetStroe，搜索商品，购买商品”这一用例。为得到最基本的测试数据，首先测试了单一用户在稳定连接环境下上述用例的 TRTOriginal 和 TRTOSF 值。前者平均值为 2.723s，后者为 3.414s。显然 SaaS 应用比普通的 Web 应用响应时间稍长，这是因为在 SaaS 应用下，用户的登录和查询操作都额外包含了租户信息的处理时间。

随后对 OSF 做了进一步的性能测试，测试对比了系统在不同用户并发数目下的性能。实验固定 5 个租户，每个租户有 $N/5$ 个并发用户(N=10,20,50,100,200)。我们使用 5 个客户端，每个客户端用多线程程序模拟并发用户，在客户端采集 TRT 数据，在服务器段采集 PT 和 MU 数据。

此外，为考虑离线访问的情况，在测试用例的最后一步，即“购买商品”中加入离线功能。首先在 β 态提交购买请求，使用“购买”的 β 实例响应该请求，随即立刻进入 α 态并使客户端 β 实例与服务器端同步。在并发环境下，查询商品和购买商品之间会产生冲突。若用户在 β 态购买成功，在 α 态同步过程中可能会因为该商品同时被其它用户购买而产生冲突，实验采用“先来先购买”的算法解决该冲突，即版本较新的 β 实例购买失败。

图 3 展示了上述实验的结果。由图 3(a)可以看出，响应时间(TRT)随用户数量的增加而增加，符合 Web 应用的特征。其中 OSF 的响应时间明显大于 Original，除了处理租户信息占用的时间外，客户端离线请求和在线同步也占用了部分时间，但性能的略微损失换来的是功能和可用性的提高。服务器 CPU 和内存使用率(PT 和 MU)随用户数量的增加而增加，使用 OSF 框架前后并无明显差异，因为离线功能并不会给服务器增加额外的运算和存储压力，服务器接收到的请求数目是相同的。

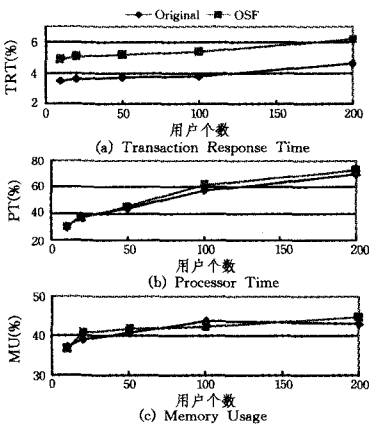

图 3　使用 OSF 前后的性能比较

图 4　查看 β 态操作和用户信息的界面

我们使用 OSF 研发了“SaaS 化项目协同管理系统”，用以对企业内各个子公司提供协同项目管理服务。项目管理中存在若干离线应用需求，如投标管理，租户在使用投标管理模块时常处于投标现场，缺少网络连接，而此时需要进行大量的数

据录入和查询工作，如投标价格、竞争单位信息等。使用OSF后的软件可以在无网络连接的环境下录入和处理招标信息，极大地改善了租户体验，并且增强了软件可用性。查看β态操作和用户信息的界面如图4所示。

结束语 本文描述了离线SaaS应用框架(OSF，Offline SaaS Framework)的构件结构，并且详细阐述了其中面向操作的构件框架的设计与实现。所有这些方法和实现都将对离线SaaS应用乃至对环境可感知的智能软件的研究和应用做出一定贡献，框架的实现还有助于提高SaaS应用开发效率，保证其可用性以及改善用户体验。进一步的工作包括在OSF中添加安全性构件、多线程优化构件，以及实现基于NoSQL数据库的面向数据的构件框架。

参 考 文 献

[1] Gruszczynski P, Osinski S, Swedrzynski A. Offline business objects: enabling data persistence for distributed desktop applications[C]// Proc. of OTM Conferences. 2005: 960-977

[2] Nguyen P, Sharma D, Tran D. Smart clients and small business model[C]// KES. 2005: 730-736

[3] Michaud P. The challenges of allowing offline usage in a SaaS based system[EB/OL]. http://www.cloudave.com/1714/the-challenges-of-allowing-offline-usage-in-a-saas-based-system, 2011

[4] Yang Y. Supporting online web-based teamwork in offline mobile mode too[C]// Proc. of WISE 2000. 2000: 486-490

[5] Ananthanarayanan G, Blagsvedt S O, Toyama K. OWeB: A framework for offline web browsing[C]// Proc. of LA-Web 2006. 2006: 15-24

[6] Ijtihadie R M, Chisaki Y, Usagawa T, et al. Offline web application and quiz synchronization for e-learning activity for mobile browser[C]// Proc. of 2010 IEEE Region 10 Conference. 2010: 2402-2405

[7] Google Gear[OL]. http://code.google.com/apis/gears, 2011-07

[8] Goncalves E E M, Leito A M. Offline execution in workflow-enabled Web applications[C]// Proc. of QUATIC 2007. 2007: 204-207

[9] JPetStroe[OL]. http://java.sun.com/ developer/technicalArticles/J2EE/petstore/ and http://www.jwebhosting.net/ jpetstore, 2011-07

[10] Cheng X, Zhang X D. Accurately modeling workload interactions for deploying prefetching in Web servers[C]// Proc. of International Conference on Parallel Processing. 2003: 427-435

(上接第98页)

间结点恢复属性哈希树根结点的计算，及利用AA的公钥进行签名验证运算。则证书验证阶段需要的时间开销最多为$K+N-1$次哈希运算及1次非对称加密运算，其中$K+N-1$次哈希运算包括计算K个隐秘特征属性的K次哈希和恢复属性哈希树根结点时不多于$N-1$个中间结点哈希值的计算。

因此，本方案计算开销除了X.509v4证书的正常签名和解签名外，仅额外增加了$(2N-1)+(K+N-1)\leqslant 4N-2$次哈希运算、$O(N\cdot\log_2 N)$次模2运算及$O(N^2\cdot\log_2 N)$次字符匹配运算。从通信复杂度分析，本方案相对于X.509v4证书仅额外增加了$N+K\leqslant 2N$个秘密参数、1个N比特二进制数和不多于$N-1$个哈希值。证书的产生、签名及合法性验证比传统的X.509v4标准所付出的额外计算是能够承受的；额外增加的通信量也是很少的。

结束语 属性证书作为权威机构认可用户所拥有属性集合的一种有效凭证，被授权管理基础设施(PMI)、信任管理及自动信任协商等领域广泛使用。现有的属性证书标准X.509V4由于直接对证书中所有的属性集合生成签名，在要求出示证书中部分属性的场合，验证方无法根据已出示的部分属性验证签名的有效性。本文提出了一种细粒度的属性证书出示方案，该方案具有与现有标准兼容、灵活性好、安全性高、网络通信负担小等优点，解决了现有研究中存在的一些问题。下一步研究还需要对X.509v4证书中部分属性移除及证书验证时所需的系统进行相应的开发。

参 考 文 献

[1] ITU-T. ISO/IEC9594-8 Rec. X.509 The Directory: Authentication Framework[S]. 2000

[2] Nykaen O. Attribute Certificate in X.509 [EB/OL]. http://www.hut.fi/~tpny-kane/netsec/final, 2000

[3] Chadwick D W, Otenko O. The PERMIS X.509 Role Based Privilege Management Infrastructure[C]// Proceedings of SACMAt'02. Monterey, California, USA, 2002

[4] Blaze M, Feigenbaum J, Lacy J. Decentralized Trust Management[C]// Proceedings of the 17th Symposium on Security and Privacy. Oakland: IEEE Computer Society Press, 1996: 164-173

[5] Winsborough W H, Seamons K E, Jones V E. Automated Trust Negotiation[C]// Proceedings of DARPA Information Survivability Conference and Exposition. IEEE Press, 2000: 88-102

[6] Persiano P, Visconti I. User Privacy Issues Regarding Certificates and the TLS Protocol[C]// Proceedings of 7th ACM Conference of Computer and Communications Security Athens. Greece, November 2000

[7] 廖俊国，洪帆，李俊，等. 在信任协商中保密证书的敏感属性[J]. 通信学报，2008，29(6)：20-25

[8] 廖俊国，凌乐真，朱彬. 一种灵活实用的数字证书中敏感属性保密方案[J]. 计算机科学，2010，37(6)：128-131

[9] 龚俭，吴桦，杨望. 计算机网络安全导论[M]. 南京：东南大学出版社，2007

[10] 肖淑婷，吴国新，孙啸寅. 支持属性选择性披露的ATN证书描述方案[J]. 计算机工程，2010，36(9)：142-144

[11] 陈性元，杨艳，任志宇. 网络安全通信协议[M]. 北京：高等教育出版社，2008：195-198

[12] Schneier B. 应用密码学(第二版)[M]. 北京：机械工业出版社，2000

[13] Pedersen T. Non-interactive and Information theoretic secure verifiable secret sharing[C]// Proceedings of CRYPTO'91. volume 576 of Lecture Notes in Computer Science, Springer, 1991: 129-140

[14] Merkle R. Protocols for Public Key Cryptosystems[C]// Proceeding of the IEEE Symposium on Research in Security and Privacy. Oakland, California, April 1980

[15] 严蔚敏，吴伟民. 数据结构(C语言版)[M]. 北京：清华大学出版社，1997：123-124