Web 聚合应用的安全跨域通信机制

孙建华，刘志容，陈浩
(湖南大学 信息科学与工程学院，湖南 长沙 410082)

摘 要：针对聚合应用现有的多种跨域通信方案难以同时兼顾安全性和高效性，提出适合聚合应用的安全跨域通信(SCDC，secure cross-domain communication)系统，将不同信任域内容封装成安全组件，借助于分层通信栈实现域间通信，并通过封装对象实现细粒度对象共享。保障了聚合应用与组件间的安全跨域通信与对象共享，且不需要浏览器做任何修改。实验表明，系统引入了有限的开销，而通信效率提高了 5 倍以上。

关键词：Web 聚合应用；组件；跨域通信；共享对象
中图分类号：TP393 **文献标识码**：A **文章编号**：1000-436X(2012)06-0019-11

Secure cross-domain communication mechanism for Web mashups

SUN Jian-hua, LIU Zhi-rong, CHEN Hao
(School of Information Science and Engineering, Hunan University, Changsha 410082, China)

Abstract: Many methods were used in cross-domain communication, whereas they were hardly to meet the security and high performance requirements. To this end, a secure cross-domain communication (SCDC) mechanism was proposed for Web mashups. It encapsulates content from different trust domains as secure components, achieves cross-domain communication with layered communication stack, and shares fine-grained objects by wrapping objects. SCDC mechanism supports secure cross-domain communication, shares objects between mashups and components without any browser modifications. Experiments show that the mechanism improves the communication efficiency more than five-fold, and only incurs limited overhead.

Key words: Web mashups; components; cross-domain communication; shared object

1 引言

在传统的 Web1.0 应用程序中，每个 Web 站点相互隔离，用户访问 Web 站点仅能得到来自本站点的信息。即使需要访问其他站点，也是通过编辑拷贝已存储在本地的信息或者用户调换网站地址的方式来访问内容，而不是直接访问信息源数据。在新的 Web2.0 潮流之下，希望网站之间打破隔离进行数据融合，使之能够共享信息。在这种背景下，聚合型网站设计方式应运而生，这就是新型的 Web 应用程序——聚合应用(mashup application)。Mashup 这个名词来源于流行音乐，将不同风格的音乐拼接，混杂在一起，构成自己独特的新曲子；Mashup 在 Web 环境中代表着整合不同来源的内容以提供交互式体验的 Web 站点或应用程序[1]。利用其他网站开放应用接口所提供的内容进行混搭，从而创造出独特的、具有新价值的 Web 应用。

Mashup 把不同源或站点的信息进行融合以保证信息共享，打破站点之间“孤岛林立”的现状，由此浏览 Web 站点更加直观便捷，用户体验更加丰

收稿日期：2011-01-24；**修回日期**：2011-08-21
基金项目：国家自然科学基金资助项目 (60803130, 61173166)；中央高校基本科研业务基金资助项目
Foundation Items: The National Natural Science Foundation of China (60803130, 61173166); The Fundamental Research Funds for the Central Universities

富。典型的 Mashup 应用当属 Housingmap，它将第三方网站 Craigslist 的租房信息和 Google Maps 提供的地图信息有机地组织起来，让租房的信息直观显示在地图上，创造出一个崭新的、互动性强的房屋搜寻站点[2]。根据以上实例背景，本文建立一个 Housingmap 聚合应用模型，由电子地图与租房信息 2 部分组成，如图 1 所示。

图 1 Housingmap 聚合应用模型

传统的 Web 浏览器安全机制遵守同源策略(SOP, same-origin policy)。同源策略规定 JavaScript (JS)代码只能访问其来自同源服务器上的数据，把来自不同源的内容相互分离。这种策略给 JS 提供安全保障的同时限制了基于 JS 的跨域访问。

Mashup 是对多个站点资源的优化组合，需要从多个分散的站点获取信息源，而不要求各个站点之间相互信任。传统同源策略过渡的安全设计没有考虑多个站点之间交互时整体的快速通信需求，也没考虑新的信任模型下的安全问题。如图 1 所示，电子地图与租房信息的数据分别来自 www.map-site.com 和 www.housing-site.com 2 个不同源的网站，在现有的同源策略下是不能够相互访问和通信。怎样整合相互独立的第三方数据、实现不同源数据之间的通信是聚合应用需要解决的问题。

另外，恶意攻击者可能重写来自其他源的网页属性，例如来自 www.map-site.com 的脚本可能重写 www.housing-site.com 网页的 location 属性，操纵原页面跳转到某个恶意的页面，而导致框架钓鱼攻击。如何保证不同源数据间通信的安全性与完整性是聚合应用需要解决的另外一个问题。

针对上述问题，研究人员在聚合应用跨域交互方面做了大量研究，其中，文献[3]利用内嵌框架(iframe)实现客户端通信。这种方法的缺点是不但很难保证消息的完整性，而且攻击者很容易进行框架钓鱼攻击。文献[4]提出了一种跨域通信方案，兼容主流浏览器。聚合应用能与一个或多个网络服务应用交互，交互过程包括 2 个阶段：设置阶段(建立中间帧，非信任帧，传输 JS 通信对象)与数据交换阶段。文献[5]提出了一种基于值的跨域通信机制，对象序列化后进行传输。其中，分批处理数据减少了整合者与小部件之间信息交换次数，由此提升了系统性能。美中不足的是，如何解决跨域通信带来的安全问题以及如何共享 JS 对象还有待进一步研究。

本文在研究相关工作的基础上提出了一种适合于聚合应用的安全跨域通信(SCDC， secure cross-domain communication)系统。首先，从各个不同的域入手，同一信任域的内容封装为组件表示，建立聚合应用的安全组件；其次，利用分层通信栈实现域间通信，即聚合应用与组件间通信；最后，利用封装技术安全实现组件间细粒度对象共享。该方案具有安全可靠性强、效率高、支持对象共享的特点，旨在为聚合应用提供一种安全可靠的跨域通信系统。

2 相关跨域通信方案

跨域，简单的理解就是在 JS 同源策略的限制，a.com 域名下的 JS 无法操作 b.com 或是 mail.a.com 域名下的对象。跨域通信面临着许多挑战，下面分别介绍 3 种传统处理跨域的方案：服务器端代理、动态创建脚本、段标识通信。

跨域请求发送至本地服务器端，然后由服务器端代理请求相应的数据发送到浏览器端。在服务器端代理的帮助下，浏览器端所有请求发送至同一域，实现了跨域通信。此方案适用于几乎所有的跨域访问，支持各种类型格式的数据，但是经过了中间代理，延迟高、开发工作量大，并且加重了本域服务器的负荷。

至于动态创建脚本方案，虽然浏览器禁止跨域访问，但仍可以引用其他域的 JS 文件，由此能够通过创建 script 节点完全实现跨域通信。该方案实现非常简单，但返回的数据格式只能是 JSON 数据，对于其他格式的数据无能为力。

最常见的跨域通信解决方案是利用 URL 段标识符(fragment identifier)[6]交换信息，传输的数据一旦超过浏览器对 URL 长度的限制则必须对数据进行分段传输。这种方法的优势在于同一页面浏览不同分段时不必刷新整个页面。不足之处在于不同浏览器 URL 长度限制有所不同，数据容量都是有限的。数据直接暴露在 URL 中，一些浏览器会删除段标识符，不能保证应用程序的一致性与可靠性。

面对复杂的聚合应用环境，以上这些方案或多或少存在下列问题：首先是开销问题，服务器代理方案经过了中间代理，开发工作量大，因此并不适合于大规模的 Mashup 应用；其次是灵敏度问题，段标识通信机制中数据容量都是有限的，消息一旦超过当前浏览器最大传输限制则必须分段进行传输。在动态的 Mashup 网络环境下，通过轮询 URL 探测分段是否变化，响应时间往往不确定，会有一定的延迟；最后一个问题是安全性，无论是动态创建脚本还是段标识通信，消息内容或者不安全或者不是预期的。Mashup 应用的通信环境往往是不完全受信任的环境，如不采取特别的安全防范措施容易导致恶意攻击或私有信息泄漏。

随着浏览器跨域通信的需求越来越强烈，HTML 标准的下一个版本 HTML5 新增了跨域通信功能，即提出了跨文档通信(cross document communication)机制。在希望发送消息的窗口或内嵌框架中调用 postMessage 方法，接收方设置一个事件处理函数来接收发送过来的消息[7]。为了解决上述问题，本文先将不同源网站视为独立的信任域，并封装为可相互直接交互的内部组件，利用 HTML5 新增的跨文档通信机制，实现组件间的通信。这种方案主要有如下优势：①安全系数高，该方案适用于不完全受信任的环境，开发人员可以选择性接收来自域的消息，如果消息的内容不安全或者不是预期的，则可以放弃相应的消息；②可靠性强，与其他跨域通信方案不同，此方案以一致的方式处理未丢弃的消息；③性能提升，此方案允许双向通信，具有安全、简单、快速的特点，不依赖于代理帧或其他页面，也不需要轮询 URL 查找数据。

3　SCDC 系统设计与实现

3.1　系统概况

本文系统为聚合应用及其各个组件(component)实现了 SCDM 系统的 JS 库，解决了跨域通信与域间对象共享的难题。SCDM 系统把不同信任域的内容封装到不同组件，且每个组件有其输入输出端口，为组件通信提供了接口；事件中心维护系统通信，通过建立分层通信栈来实现组件间的通信，跨域通信依赖于跨文档通信机制；对象共享是聚合应用域间通信的高级应用。

系统支持多个较新版本的浏览器，Internet Explorer 8.0+(IE8.0 及以上版本)，Firefox 3.1+，Safari 4.0+，Opera 9.5+，Google Chrome 1.0+。对于不同的浏览器，实现代码大体上类似。

图 2 以 2 个组件之间的通信为例说明系统的整体框架，多个组件之间通信时架构与此类似。SCDC 系统包含 3 个主要模块：域封装模块、域间通信模块、细粒度对象共享模块。

域封装模块是系统的入口和核心。聚合应用由不同源(域)的内容组成。鉴于安全性考虑，不同信任域的内容封装到不同组件中以起到域间隔离的作用，并且组件的输入输出端口为域间通信模块提供接口。

域间通信模块为组件模块和对象共享模块提供服务。判别通信双方是否处于同一个域，完成域间通信识别的任务，并且为不同域之间通信搭起桥梁，完成系统跨域通信的任务。

细粒度对象共享模块是系统的高级应用模块。目前组件间并不能直接跨域共享对象，但是借助基于值传输的跨文档通信机制能在对象序列化为字符串后能进行对象的共享。为了更好地安全地实现信息共享，本模块根据域封装模块所提供的组件，及域间通信模块提供的服务获取通信状态，实现组件间细粒度的对象共享。细粒度对象共享的过程是先建立封装对象，并且由策略系统明确定义策略以控制对象共享，然后对象序列化后通过域间通信模块进行传输。

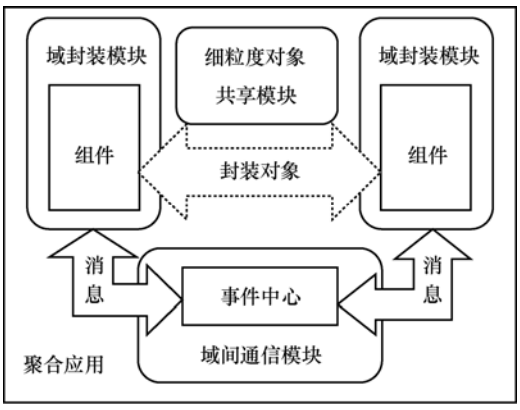


图 2　系统框架

3.2　域封装模块

信任域(trust domain)[8]可认为是处于聚合应用完全控制下的安全域(域指的是 DNS 域或 IP 地址)。信任域与 DNS 域有着密切联系，前者在后者的基础上加入了安全防范措施。

组件可以定义为同一信任域内容的封装。组件

源由第三方提供，每个组件逻辑上相互独立。组件拥有独特的输入端口和输出端口，为后面的异步传输消息垫定基础。

为了安全性考虑，系统需要隔离各个组件以防止恶意攻击。解决方法是将不同的组件置入不同的内嵌框架 iframe 中以达到隔离目的，任意组件不能改变 domain 属性以攻击其他组件。对于上例，Housingmap 聚合应用模型组合了来自 www.map-site.com 的电子地图与 www.housing- site.com 的租房信息，分别把这 2 个来自不同信任域的内容封装为电子地图组件与租房信息组件。然后，电子地图组件与租房信息组件分别置入各自的 iframe 起到域间隔离的作用。

对于来自同一服务器的数据，但属于不同信任域，也可以建立多重 DNS 域从而保证组件间隔离。例如，对于来自服务器 www.map-site.com 与信任域为 t1、t2 的数据，能够建立 2 个 DNS 域 t1.map-site.com 和 t2.map-site.com，分别封装到不同的组件，为下文的域间通信打下基础。

3.3　域间通信模块

域间通信模块负责各个信任域之间的通信。该模块借鉴与结合了浏览器安全模型的实现方法，在组件端口与事件通道之间通过发送/接收消息方式实现聚合应用与组件间的内容交互。

事件中心(event hub)由仲裁域间通信的事件通道(channel)组成，是维护系统通信的核心部件。事件中心实际上是在多对多通道上发送与接收消息的 pub/sub 系统，但又不同于传统的 pub/sub 系统。事件中心管理连接组件端口与事件通道之间的通信，组件端口与事件通道的命名空间相互隔离，由此避免了组件端口的冲突，提高了组件的重用性。组件有其输入端口(读端口)与输出端口(写端口)，从组件的输出端口发送消息到与之相连的事件通道上；组件的输入端口从与之相连的事件通道接收消息，由此构成了组件与聚合应用，以及组件与组件之间的通信。如图 3 所示，组件 1 发送消息到事件通道 1、3，组件 1 从事件通道 3 接收到消息。在一般情况下，组件可发送消息到多个事件通道，同时也可从多个事件通道接收消息。

如何保证组件之间通信顺畅，文献[8]中采用的方法是借助于 URL 段标识符实现域间通信。消息一旦超过当前浏览器最大传输限制则必须分段进行传输，直到上一分段读取完毕后才能读取下一个分段。

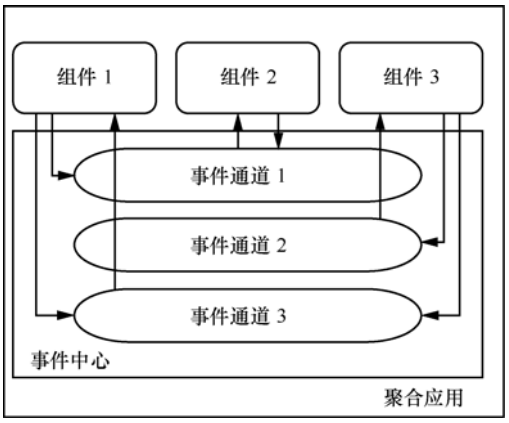


图 3　简化通信

本文的解决思路是通过跨文档通信机制实现聚合应用与组件的域间通信。对于聚合应用，组件在域封装模块已置入 iframe 帧中，以组件帧表示。而且每个组件均包括隐藏的通道帧，通道帧与聚合应用主页面(www.mashup.com)属于同一个域，因此通道帧与聚合应用主页面实现域内同步通信。而通道帧与组件帧通过跨文档通信机制实现域间通信。

具体实现架构利用 JS 库底层实现聚合应用与组件的域间通信，实现架构类似分层通信栈，由事件中心层、事件通信层及跨文档通信层 3 部分组成，在组件及聚合应用中的同等层使用较低层进行通信。如图 4 所示，事件中心层管理组件端口与事件通道，对于聚合应用，事件中心层的主要任务是装载组件与卸载组件，建立事件通道与删除事件通道，

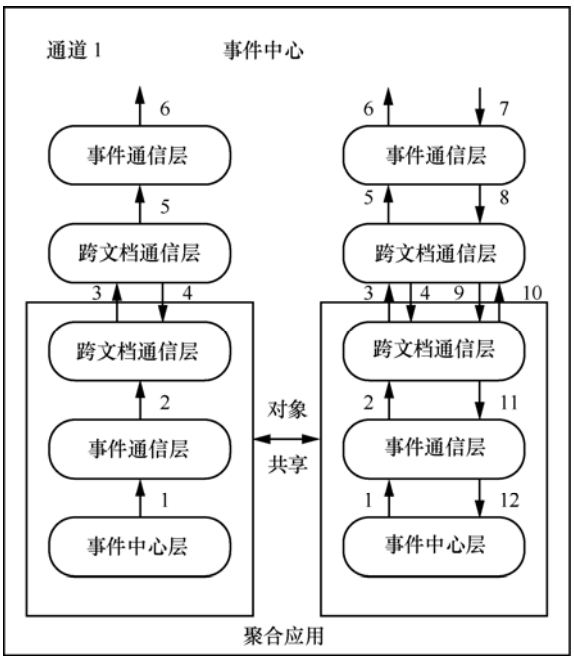


图 4　分层通信栈

连接组件端口与事件通道等；对于组件，这一层负责处理该组件各端口发送与接收的事件。事件通信层负责多路传送多个组件端口的消息。跨文档通信层是为聚合应用与组件实现跨域通信，也就是通道帧与组件帧的通信过程。

下面列出实现域间通信的常用 API，如表 1 所示，重点描述下组件状态的生命周期，参照 getComponentState 函数。从 loaded 到 wired 状态转换由聚合应用控制，意味着组件能够开始发送消息；从 wired 到 startedCleanup 状态也是由聚合应用所控制，意味着开始卸载组件；而从 startedCleanup 到 doneCleanup 状态则是由于超时或组件处理不当引发的，这个过程由组件所控制；剩下的状态转换则是由系统自身初始化的。为了避免轮询组件状态，在组件及聚合应用状态转换时注册监听函数。

组件与聚合应用都通过发送与接收消息方式进行通信。用一个简单的例子描述利用分层通信栈的通信过程。首先，聚合应用主页面中装载 2 个组件，初始化通信，发送消息 message 给组件，主要调用如下 API：

1) seHub=new SEHub(channelListener, stateTransionListener);

// 建立事件中心，通过 channelListener，stateTransitionListener

监听事件通道及状态转换 (addReader、addWriter、componentWired 等)

2) seHub.loadComponent("c1",["port0"], ["port0"]); //装载组件 1

3) seHub.loadComponent("c2",["port0"],["port0"]); //装载组件 2

4) seHub.createChannel("topic1"); //建立事件通道 topic1

5) seHub.publishOnChannel("topic1", message);

//发送消息 message 给组件

组件初始化后，发送消息到事件通道，主要调用如下 API：

1) seHub=new SEHubClient(stateTransitionListener); //组件方事件中心层，通过 stateTransitionListener 监听组件状态

2) seHub.registerCallback(port, seHubCallback1);

//通过 seHubCallback1 回调函数显示组件端口的消息

表 1　　域间通信常用 API

API	语法	解释
聚合应用 API	loadComponent (componentId,inPortNames,outPortNames)	装载组件
	deleteComponent (componentId)	删除组件
	createChannel (channelName)	建立事件通道
	deleteChannel (channelName)	删除事件通道
	addReader (channelName, componentId, inPortName)	给事件通道增加一读端口
	addWriter (channelName, componentId, outPortName)	给事件通道增加一写端口
	removeReader (channelName, componentId)	从事件通道删除一读端口
	removeWriter (channelName, componentId)	从事件通道删除一写端口
	getComponentState (componentId)	获取组件状态：start→loaded→wired→startedCleanup→doneCleanup→unloaded
	publish (componentId, port, message)	从组件输出端口发送消息到事件通道
	publishOnChannel (channelName, message)	组件输入端口从事件通道接收消息
	distribute (topic, message)	发送某通道的消息到对方组件
	messageRecieved (message)	处理收到的消息
组件 API	publish (port, message)	组件的事件中心层从输出端口发送消息
	publish (topic, message)	组件的事件通信层发送消息到某通道
	send (message)	跨文档通信层发送消息给对方帧
	messageRecieved (message)	处理收到的消息
	setRecieved (callback)	为收到的消息设置回调函数
	registerCallback (port, callback)	为特定端口注册回调函数

3) seHub.publish(port, message);

//发送消息 message 给事件通道

聚合应用发送消息给组件的通信过程(组件发送消息给聚合应用的通信过程与之类似)简单概括如下(步骤如图 4 所示)：聚合应用中事件中心层 SeHub(publishOnChannel)发送消息到与该事件通道相连的组件输入端口(第 7 步)，事件通信层 SEComm (distribute)多路传送消息给相关组件(第 8 步)，跨文档通信层 CommLib(send)通过跨文档通信机制发送消息给组件帧，轮询监听，当检测到有新消息时调用跨文档通信层(messageRecieved) 处理收到的消息(第 9、10 步)，组件的事件通信层 SECommClient (messageRecieved)处理从跨文档通信层收到的输入数据(第 11 步)，调用 SEHubClient (messageRecieved)处理从事件通信层收到的消息(第 12 步)，最后调用组件注册的回调函数 seHubCallback1 显示组件端口及收到的消息。

3.4　细粒度对象共享模块

Web 主体间共享资源通常有 2 种处理方案：共享所有资源，或者相互隔离，实现零共享[9]。通常各个主体共享所有资源容易导致跨站脚本攻击(XSS)，由此研究人员提出了多种方法来隔离各个主体。进一步，如何在各个隔离的主体间共享部分对象，针对这个问题，提出了细粒度共享对象策略。

首先，利用 SCDC 系统的 JS 库中对象封装策略，建立与原对象一致的、可控制的封装对象 view[10]。一致性表示封装对象 view 可以取代原对象进行访问；可控制性意味着可以定义策略限制接收方访问共享对象的某些属性或方法。其次，建立细粒度策略(基于通知的策略)约束访问共享对象。细粒度策略由方面系统(aspect system)实现，能控制对象的行为。

如图 5 所示，为对象 map 建立封装对象 map_control. view 来取代原对象的访问，并通过为原对象的属性与方法引入 getters、setters、caller 访问器利用 map_control. definePolicy 设置策略以控制访问封装对象 view。view 可以看作是代理与策略的组合体，代理相当于一个封装器，策略即为一个函数。为每个不同的对象建立一个新的封装对象。同样地，对于一个函数对象，相应地定义一个新的封装函数以取代对原函数的访问，并为其定义策略函数。

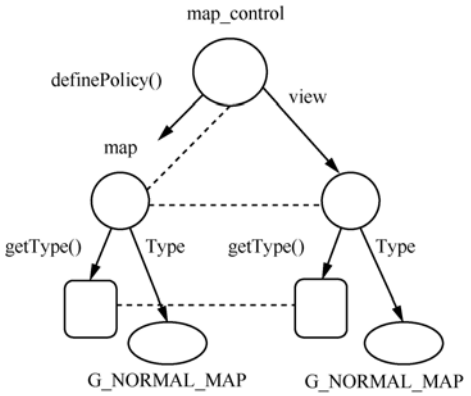


图 5　封装 map 对象

1) var map = {getType: function (v) {}, Type: G_NORMAL_MAP};　//定义地图类型对象，包括地图类型属性，及返回当前的地图类型

2) var map_control = makeView(map);　///封装对象

3) var permitGet = function (obj, prop) {return obj[prop];};

//定义 getters 的通知函数

4) var permitCall = function (f, t, a) {return f.apply(t, a);};

//定义 caller 的通知函数

5) map_control. setPolicy(map, {getters: {getType: permitGet, Type: permitGet}});　//定义策略

6) map_control. setPolicy(map.getType, {caller: permitCall});　//定义策略

7) peercomponent. send(map_control.view);

//把 view 对象发送给对方组件

对于 Housingmap 聚合应用模型，细粒度的共享策略只允许该应用与第三方地图服务共享与地图相关的页面，双方能够在保持各自内部不变量的前提下交互 JS 对象、方法等。如图 5 所示，电子地图组件与租房信息组件共享某一对象，如地图类型(包括普通地图、卫星地图、地形地图等类型)。首先，电子地图组件建立其封装对象 view (map_control.view)以代替原对象。然后通过定义策略控制对封装对象的访问，如定义策略即编码白名单，租房信息组件只能读其属性 Type，调用方法 getType，但不能重定义方法或属性，也不能操作电子地图组件的其他对象。

建立封装对象 view 后，怎样实现组件间 view 对象的共享。针对这一问题，浏览器并未为跨域传

送消息设定特定的事件通道，但能够借助于跨文档通信机制与分层通信栈结构，对象能够序列化后实现共享传输。如图 6 所示，组件 1 与组件 2 共享封装对象时，利用 SCDC 的 JS 库发送方序列化对象后通过分层通信栈发送给对方；接收方收到消息后把字符串反序列化为对象。由此，当租房信息组件要请求 map_control.view 的属性 Type 时，首先发送请求给电子地图组件，依据 view 策略得到结果，然后返回给租房信息组件。

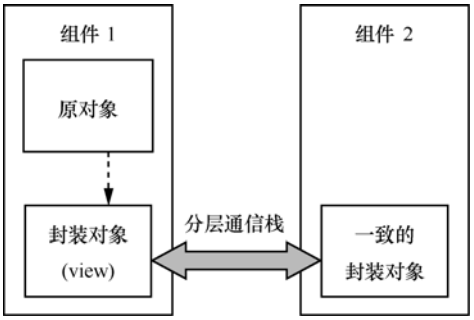


图 6　封装对象 view 共享过程示例

细粒度对象共享一定程度上解决了对象传输的安全性，同时也引入了一些新的问题，如异步问题。

1) 封送处理库负责序列化与反序列化对象，并对基本的 JS 操作如调用函数、读写属性、抛出异常等进行译码。如果组件 1 远程共享一个表格对象 table 给组件 2，组件 2 请求封送处理库获取 table.parentNode.parentNode.parentNode.cookie 的值，这将泄露了文档的 cookie 值；如果组件 1 远程共享其封装对象 table_control. view，view 能执行策略代码仅能访问 table 对象的白名单属性或方法。封装对象机制限制访问对象，解决了远程对象共享带来的部分安全问题。

2) 浏览器的同源策略规定每个框架有独自的类型链与全局对象集合，一定程度上确保了框架间的隔离。由于聚合应用需要跨域访问对象，同源策略受到限制。SCDC 系统的细粒度对象共享模块在访问对象之前进行检查，以确认是发送方对象的封装对象 view 或接收方对象的代理。如果 get、set、call 等触发某个对象既不是 veiw 也不是代理，则拒绝此对象，进一步保证对象传输的安全性。

3) 由于跨文档通信机制本身的异步性，通过分层通信栈实现的远程对象共享引入了异步问题。如图 7 所示，图 7(a)中转化为远程对象的封装对象 view，调用 2 个“.”则是异步调用。这就需要转化代码结构，调用者必须提供一个回调函数以保持传输的完整性与连续性，这就是连续传递模式(CPS，continuation passing style)。如图 7(c)所示，客户端自动全局转化为 CPS 形式进行交互，在调用 *y*()与 *z*()时，代理为了保持连续性注册回调函数以控制当前线程。

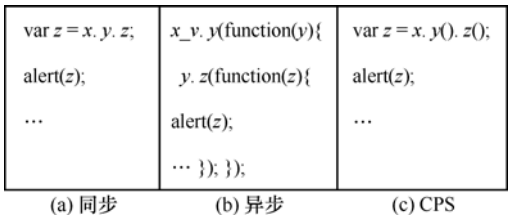


图 7　共享对象交互方式

4　安全策略

4.1　SCDC 系统整体安全策略

上文已经提到组件隔离及组件间通信，至于哪些组件之间可以交互，哪些组件之间禁止交互，则由 SCDC 系统的安全策略所定义。具体的策略来自多种高级策略，例如企业级聚合应用，安全策略则可能是企业级策略、部门策略、终端用户策略的组合。一方面，组件提供方明确它们的组件怎样整合到聚合应用中；另一方面，聚合应用提供方可能不同等看待各个组件，可能会阻止部分组件相互交互。总之，安全策略依赖于具体的实体应用程序，包括组件间访问控制策略与组件到服务器端的访问控制策略。

组件间访问控制策略包括建立与删除组件、建立与删除事件通道、哪个组件能在某个事件通道上发送或接收事件等。事件中心负责执行策略，一种方式可以在聚合应用的代码中明确定义策略；另一种方式为事件中心定义一个配置策略文件，可以从中读取安全策略，这样分离了策略定义与聚合应用中真正的代码实现。

组件到服务器端的访问控制策略包括认证组件与受限访问权限。在组件装载之前，必须先核实认证，并且根据组件所在域确保组件装载到合适的 iframe 帧中，使其不易受到人为攻击。这种方式不管是客户端聚合应用还是服务器端聚合应用都有统一的访问控制策略。

上述定义了 SCDC 系统的整体安全策略，但是跨域通信容易导致框架钓鱼攻击，共享信息容易泄露个人隐私信息，针对这 2 个问题本文特别加入了

针对性的安全策略。

4.2 防范框架钓鱼攻击

框架钓鱼攻击，对于利用 iframe 来隔离不同信任内容的网站来说是很常见的漏洞，也是比较严重的攻击类型[11]。它一定程度上破坏了应用程序的完整性，但并不意味着应用程序被恶意组件所控制。框架钓鱼攻击还能窃取输入的信息，包括个人认证信息或密码等。在聚合应用中，这种攻击不仅能改变组件帧的 location 属性，也能作用于事件通道帧与聚合应用主页面之上。

对于聚合应用，组件、事件通道、聚合应用主页面三者易发生框架钓鱼攻击。由于 URL 栏仅能提供微弱的钓鱼保护能力，而且不像传统的钓鱼攻击，聚合应用中框架钓鱼攻击的时机是任意的。本文在各个页面中利用 onunload 事件处理器，超时设置及通道帧三者相结合共同防范框架钓鱼攻击。

首先，当组件受到攻击时触发 onunload 事件并发送通知消息给聚合应用，然而组件与聚合应用通过事件通道进行异步通信，不能保证在组件卸载完之前通知消息传送完毕。换个角度考虑，替换某个组件时将会触发通道帧的 onunload 事件，通过调用 JS 函数通道帧与聚合应用实现同步通信，因此在 onunload 事件完成之前必能成功通知聚合应用。根据上述分析得知，可以用统一的方法来处理针对组件与事件通道的攻击。

其次，另一种可能的情况是在通道帧装载完毕之前取代组件，这时在聚合应用中设置一个超时处理成功初始化事件通道通信。一旦超时，触发错误处理程序以决定是卸载或重载组件。利用事件通道的 onunload 事件处理程序面临一个挑战，当用户操作离开聚合应用页面时，该事件也被触发。为了避免错误判断，可以延迟计时器的警告时间，如果用户仍停留在聚合应用中，超时则通知聚合应用存在潜在的框架钓鱼攻击。

最后，检测聚合应用主页面的框架钓鱼攻击。有时很难区分是框架钓鱼攻击还是用户任意操作离开这个网页。针对这个问题，设计一个警告框来通知用户 URL 的变化，没有浏览器的支持很难区分上述 2 种情形。虽然这种方法对系统性能有一定的影响，但是这是唯一一种确保用户安全性的方法。

4.3 对象共享安全性

随着计算系统的不断发展，资源共享变得越来越重要，而由此导致的安全问题也变得不容忽视。SCDC 系统的对象共享模块容易泄露私有信息，所以做好安全防范工作显得尤为重要[12]。

为了保证私有信息不被泄露，递归封装对象是理想的选择。递归封装即为某个对象的所有属性（包括继承的属性）及返回值定义访问器或代理函数，并且建立封装对象须遵循引用一致性策略。引用一致性指以对象 ID 为索引存储一个原对象与封装对象的关联表，如果请求的对象不在表中则生成一个新的封装对象，否则返回同一个对象以保持引用对象的一致性。

同时，双重的输入输出封装构筑另一道安全屏障。除了不允许原对象引用泄漏(exporting)外，同时不允许不受信任的代码进入(importing)访问。输入封装是为第三方对象(参数和重定义的属性)设计的，即当一个封装对象的某个方法接收参数或某个属性重赋值时都必须进行输入封装。下面举个实例说明怎样保证对象共享安全性，例如定义对象 obj，包含一个属性 prop 与一种方法 proc，其中方法 proc 可读写，属性 prop 则不可读写。

1) var obj = {proc: function () {}, prop: “message”};

2) var a = makeView(obj);

3) var permitSet = function (o, p, value) {o[p]=value;};

4) a.definePolicy(obj, {getters:{proc: permitGet}, setters: {proc: permitSet}});

5) peercomponent. send(a.view); //把 a.view 发送给对方

建立对象 obj 的封装对象后，发送给接收方，潜在的攻击如下：

//obj_v 是接收方的对象 obj 的封装对象

1) obj_v. proc = function(obj1) {broadcast(obj1. prop); };

2) a.definePolicy(obj. proc, {funCall: permitCall});

//调用原对象的 proc 方法

3) obj. proc(obj);

一种攻击方式是 view 接收方重定义方法 proc，从而泄漏属性 prop 的值；另一种攻击方式是当对象 obj 调用 proc 方法时，新的 y 函数则会广播属性 prop 的值。怎么利用输入与输出封装来防范这种攻击呢？首先，obj_w 定义为 obj 的输出封装器，任何

尝试设置 obj_w 的 proc 方法时将受到约束限制；其次，setters 代理检查赋值号右边的值未被封装，则在赋值之前为不被信任的函数进行输入封装；最后 obj.proc 的输入封装器发现它的参数处于受保护状态，则参数必须封装后才能实现输出，由此 prop 不再是输出封装器的属性。

5　实验

实验平台：全部实验在频率为 1.5GHz 的 Intel Celeron M 的处理器，512MB 内存的机器上实现的，所用的操作系统是 Windows XP，采用服务器 Apache Tomcat 5.5 发送数据到浏览器(Internet Explorer 8.0，Firefox 3.6.3，Safari 4.0.5，Opera 10.54，Google Chrome 6.0.401.1)。服务器端代理需要修改服务器，对服务器不透明；动态创建 script 节点，需要引用其他域的 JS 文件且支持的数据格式有限；而段标识通信机制是传统跨域通信方案中最常见的通信方案，应用广泛，对服务器透明且不需引入其他 JS 文件等，可比性强，所以本文选择采用段标识跨域通信机制的系统(FIC 系统)与本文系统进行测试，对比了两者在数据吞吐率(data throughput)和事件发生率(event rate)以及组件装载延迟(component load latency) 3 方面的实验数据，同时还测试比较了 SCDC 系统支持对象共享前后的 JS 执行时间开销。

5.1　数据吞吐率

组件数从 1、2 递增到 32 个，轮询时间间隔取 10ms、20ms、40ms、80ms 不等，测试数据吞吐率(横纵坐标为对数刻度)。开始从聚合应用传输 4KB，8KB 等小数据量到各个组件中，耗时非常短。然后传送 1MB 的数据，对比系统所用的时间。所有的实验数据都是测验 5 次后取平均值得到的，减少了随机因素的影响。实验结果表明随着组件数目的增加，SCDC 系统的吞吐率随之增长，只是组件数越大，吞吐率增长幅度越小些。由于篇幅限制，在此只给出 Firefox、Google Chrome 2 个常用浏览器的测试结果，如图 8(a)和图 8(b)所示。

同时对比 SCDC 系统与 FIC 系统，比较图 8(a)与图 8(c)，实验结果显示 SCDC 系统比 FIC 系统的吞吐率高出 5 倍之多。主要原因在于 FIC 系统受到浏览器 URL 长度的限制，当传输数据量比较大时，则要分段进行传输，而 SCDC 系统则无此限制，由此数据吞吐率明显提高。

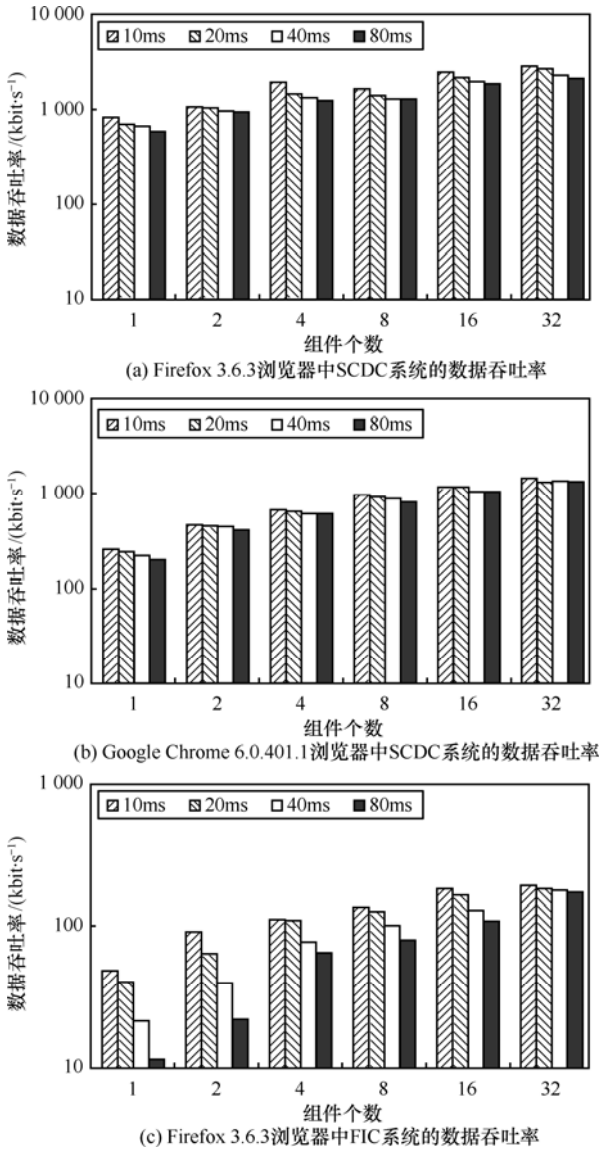


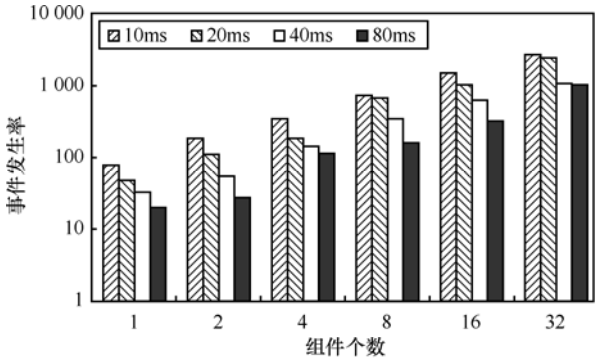
图 8　数据吞吐率比较

5.2　事件发生率

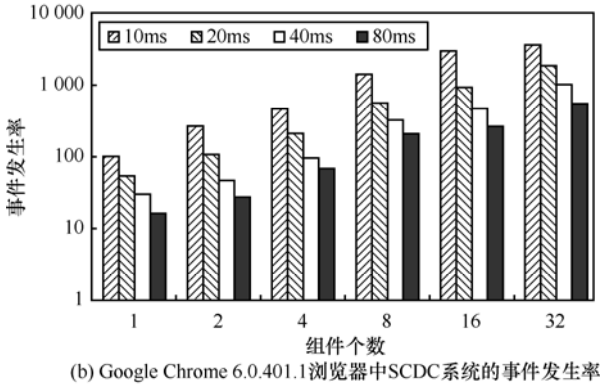
事件发生率，衡量对于小事件，系统所能支持的最大事件率。对于许多聚合应用来说，组件之间需要交换小事件以响应用户行为。事件发生率系统传输 15 个字符的小负载(简单的名/值对)进行测试事件发生率。测试结果如图 9(a)和图 9(b)所示，对于各个浏览器，存在一定的差异，随着组件的数量，轮询时间间隔的变化，事件发生率呈现上升的增长趋势，主要缘于跨文档通信机制是异步传输的机制。同时对比 SCDC 系统与 FIC 系统，实验结果显示，SCDC 系统的事件发生率比 FIC 系统提高了 5 倍左右，如图 9(a)和图 9(c)

所示。

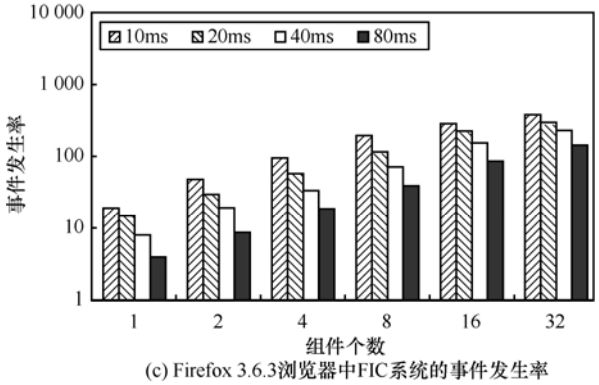
事件发生率系统传输 1MB 的大负载进行测试事件发生率。大负载测试结果如图 10 所示，比较图 9(a)与图 10，可以发现与小负载测试结果相似。由此可知，事件发生率与负载的大小无关。



(a) Firefox 3.6.3浏览器中SCDC系统的事件发生率



(b) Google Chrome 6.0.401.1浏览器中SCDC系统的事件发生率



(c) Firefox 3.6.3浏览器中FIC系统的事件发生率

图 9 事件发生率比较

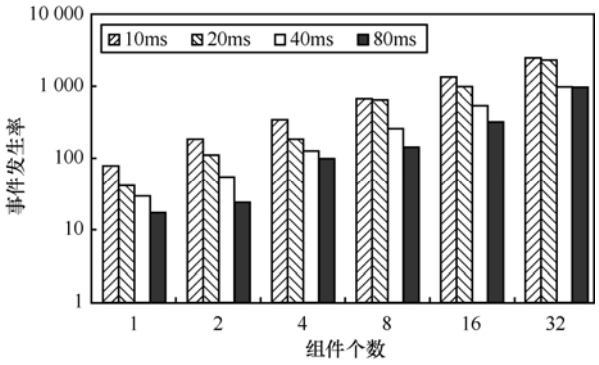
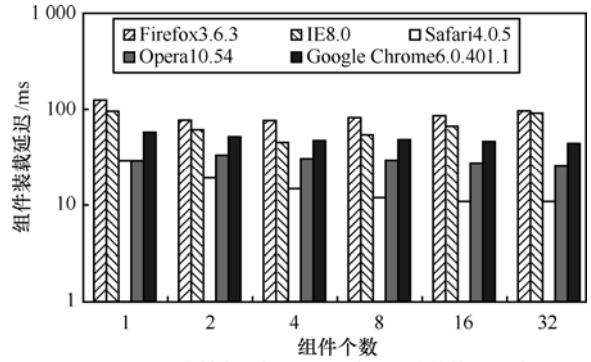


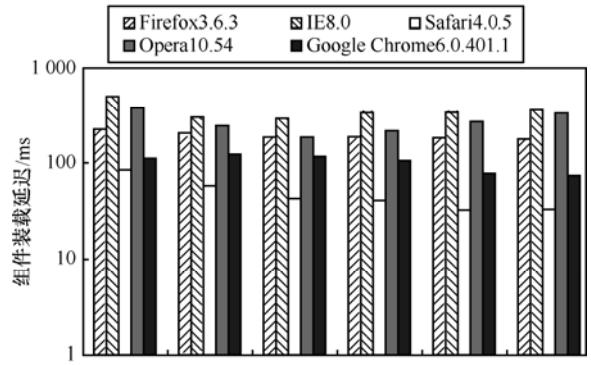
图 10　Firefox 3.6.3 浏览器中 SCDC 系统的大事件发生率

5.3　组件装载延迟

当同时装载的组件数目增加时，测试每个组件的装载延迟。总装载延迟是从聚合应用建立组件时开始，直到聚合应用收到成功装载所有组件的确认消息。如图 11 所示，随着组件数目的增加，组件装载延迟大体上是有所下降的。尤为特殊的是，Safari 浏览器使用浏览器缓存，所以延迟很小。同时，对比 SCDC 系统与 FIC 系统发现，组件装载延迟明显减少。



(a) 5种浏览器中SCDC系统的组件装载延迟对比



(b) 5种浏览器中FIC系统的组件装载延迟对比

图 11　组件装载延迟比较

5.4　共享封装对象前后的 JS 执行时间开销

比较组件之间传输封装对象前后的 JS 执行时间，实验结果显示系统开销很小。在增加支持对象共享功能后，JS 库文件大小增加了 25.1KB，从建立连接、发送请求、等待响应、接收数据总共耗时不过 16ms 左右。可以得出这样的结论：共享封装对象 view 后 JS 执行时间开销是相当小的，对 SCDC 系统性能影响甚小。

6　结束语

聚合应用组合了来自不同源的信息，由此信息的交互就成为一个瓶颈。本文在分析比较传统跨域通信方案优缺点的基础上，利用 HTML5 新增特性跨文档通信机制以及借助于分层通信栈结构实现了适合聚合应用的安全跨域通信系统；并且在系统中引入了细粒度的对象共享，通过封装对象及定义策略来约束控制对象，以代替对原对象的访问。同时，跨域通信与对象共享导致的安全问题也不容忽视，相应地增加了针对性的安全防范措施。通过一系列实验及对其结果进行比较分析可知，运用该 SCDC 系统的聚合应用程序，性能有了大幅度的提升，事件发生率、吞吐率明显较之前的通信方案提高了 5 倍以上；减少了组件装载延迟时间；同时，对象共享也仅引入很小的开销。

今后的研究工作包括在聚合应用环境下进一步完善跨域通信系统的安全性，研究更全面的安全措施。同时在对象共享方面有所突破，使得所有浏览器本身支持细粒度对象共享。

参考文献：

[1] MERRLL D. Mashups: new member of Web applications[EB/OL]. http://www.ibm.com/developerworks/cn/xml/x-mashups.html, 2008.

[2] 沈云凌. Mashup 技术的研究与应用[D]. 上海：上海交通大学, 2008. SHEN Y L. Research and Application of Mashup Technology[D]. Shanghai: Shanghai Jiaotong University, 2008.

[3] LEE G. Personal communication on XDDE[EB/OL]. http://www.openspot.com, 2007.

[4] JACKSON C, WANG H. Subspace: secure cross-domain communication for Web mashups[A]. The 16th International Conference on World Wide Web[C]. New York, USA, 2007. 611-619.

[5] BARTH A, JACKSON C, LI W. Attacks on JavaScript mashup communication[A]. IEEE Computer Security and Privacy 2009(W2SP 2009)[C]. California, USA, 2009. 323-330.

[6] THORPE D. Secure cross-domain communication in the browser[J]. The Architecture Journal, 2007, 12(6):14-18.

[7] HICKSON I. HTML5 Web messaging[EB/OL]. http://dev.w3.org/html5/ postmsg/, 2010.

[8] KEUKELAERE F, BHOLA S, STEINER M, *et al*. SMash: secure component model for cross-domain mashups on unmodified browsers[A]. The 17th International Conference on World Wide Web[C]. New York, USA, 2008. 535-544.

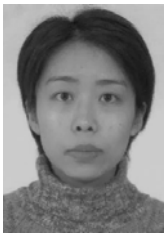
[9] BHOLA S, CHARI S, STEINER M. Least privilege 2.0: access control for Web 2.0 applications[EB/OL]. http://domino.research.ibm.com/comm/research_projects.nsf/pages/web_2.0_security.smash.html/$FILE/least-privilege_2.0.pdf, 2008.

[10] MEYEROVICH L, FELT P, MILLER M. Object views: fine-grained sharing in browsers[A]. The International Conference on World Wide Web(WWW2010)[C]. NC, USA, 2010. 67-76.

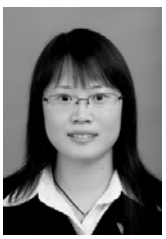
[11] BARTH A, JACKSON C, MITCHELL J. Securing frame communication in browsers[A]. The 17th USENIX Security Symposium(USENIX Security 2008)[C]. California, USA, 2008. 83-91.

[12] MEYEROVICH L, ZHU D, LIVSHITS B. Secure cooperative sharing of JavaScript, browser, and physical resources[A]. IEEE Compute Symposium on Security and Privacy 2010(W2SP 2010)[C]. California, USA, 2010. 15-19.

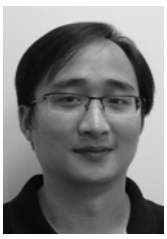
作者简介：



孙建华（1977-），女，河南焦作人，博士，湖南大学副教授，主要研究方向为网络安全。



刘志容（1985-），女，湖南耒阳人，湖南大学硕士生，主要研究方向为网络安全。



陈浩（1977-），男，湖南湘阴人，博士，湖南大学副教授，主要研究方向为网络安全和分布式计算。