OSH 可行性报告

实时文本协作系统

徐直前 PB16001828
吴永基 PB16001676
黄子昂 PB16001840
金朔苇 PB16001696

2018 年 7 月 3 日

目录

1 项目简介

本项目旨在实现一个基于 CRDT（Conflict-free Replicated Data Type）技术实现的实时文本协作系统。能够实现更加精细的权限控制，使得管理员能控制每一个用户具体能够编辑的区域，而不是现在几乎所有的文本协作系统所实现的笼统的对全文只读或者可编辑的权限控制。同时也支持 Markdown 语法，实现 Markdown 的实时预览。相比于 Word 这样的可见即可得的富文本格式，Markdown 这样的基于语法控制排版的方式更受技术人员、产品经理、科技记者、小说作者等人群的喜爱。本实时协作系统最终将以网页端的方式实现，具有良好的跨平台性，也便于用户使用。系统将使用 JavaScript 开发，使用 Node.js 作为后端，使用 npm 包管理器就可以轻松的在服务器上进行部署，移植成本也相当低。

2 项目背景

通常在软件开发中使用的多人协作解决方案，就是 Git 这种版本控制系统了。Git 这种版本控制系统通常是每个用户各自工作，完成自己的部分代码，最后再将自己的代码 merge 到项目中去。在此过程中，客户端不需要和服务器以及其他客户端进行通许。所以，这里也就不需要一个锁的机制，用户就可以做到并发地编辑。但是在实时协作系统中同步性就成为了最大的问题了。同步性

让我们来看一个例子。假设客户端 Alice、Bob 和服务器端开始的状态都是 ABCD

现在 Alice 要在 D（即第四个字符（假设 offset 从 0 开始，D 的 offset 为 3））之前插入字符 X，同时 Bob 要删除字符 B，那么由于延时的问题，那么 Alice 可能会认为插入在删除之前执行，但是服务器认为删除操作是在插入操作之前执行的。那么 Alice 看到的结果就是 ACXD，而服务器上的结果是 ACDX。这显然就出现来致命的问题，每个编辑者看到的结果不一样。显然，需要一个精心设计的机制才能解决这个问题。

我们在调研报告中已经调研过 Google Docs，Apple iWork，Microsoft Office Online、石墨文档，还有像 EtherPad、Firepad 这样的开源项目。目前我们还没有找到一个理想的方案，实现了对 Markdown 或者 Latex 等的支持，以及实现除了控制每个人对整个文档的只读和可编辑以外更精细的权限控制。此外，以上大部分产品都采用了 OT 的技术实现。CRDT 相比 OT 有更好的可延伸性，更好的容错性等诸多优点，因为其是新诞生的技术，还没有多少产品应用了 CRDT 技术。在此，我们也尝试使用 CRDT 技术来构建一个可以支持 Markdown 实时预览，能实现更加精细的权限控制的实时文本协作系统。

3 理论依据 - CRDT

3.1 简介

CRDT 是 Conflict-free Replicated Data Type 的缩写，换言之“无冲突可复制型数据”。

人类目前实际可行的构建超大规模系统的比较好的解决方案就是利用分布式文件系统。但是分布式文件系统正确性很难保证 CAP 定理证明了在构建分布式系统的时候，Consistency（一致性），Availability（可用性），Partition tolerance（分区容错性），这三者只可以同时选择两样。分区容错性是实际运营的分布式系统所必需的。所以必须在一致性和可用性中选择其一。选择一致性，构建的就是强一致性系统，比如符合 ACID 特性的数据库系统。选择可用性，构建的就是最终一致性系统。前者的特点是数据落地即是一致的，但是可用性不能时时保证，这意思就是，有时系统在忙着保证一致性，无法对外界服务。后者的特点是时时刻刻都保证可用性，用户随时都

可以访问，但是各个节点之间会存在不一致的时刻。最终一致性的系统不是不保证一致性，而是不在保证可用性和分区容错性的同时保证一致性。最终还是要在最终一致性的各节点之间处理数据，使他们达到一致。

CRDT 是一个解决这种问题的很好的方案。其思想从总体上来说就是利用多存储的信息来在某一时刻解决一致性问题。在实际系统中应用，我们必须要考虑很多的数据类型和应用场景。CRDT 就是这样一些适应于不同场景的可以保持最终一致性的数据结构的统称。举例来说，假如我们想利用分布式系统来解决一个账户的支出收入问题。假设这个账户是 T，初始化有 100 块钱，用户可以通过系统里面好几个节点，例如 A,B,C, 访问它。那么我们最终的分布式系统，都会给 A,B，C 三者权限以对账户进行操作。这时我们就需要多加入一些信息来保证信息的正确性。此时我们这样设计这样的一个系统，每个系统存储的不是一个最终的数值，而是一系列包含了时刻与余额的记录，假设我们的系统从 t0 时刻开始的，那么在我们的例子里面，t1 时刻

1. A 系统存储的是 (t0,100), (t1,110)
2. B 系统存储的是 (t0,100), (t1,90)
3. C 系统存储的是 (t0,100), (t1,100)

这样的结构使得我们在传输了足够的信息之后，都能达成一致性。例如对于 C 系统，当收到足够多的信息，即是除自己之外所有的节点信息（A 和 B) 后，就得到了这样的结论

1. A 系统在 t0 至 t1 之间产生的变化是 +10
2. B 系统在 t0 至 t1 之间产生的变化是-10
3. C 系统在 t0 至 t1 之间产生的变化是 +0
4. A 和 B 系统与 C 在 t0 时数据一致，在 t1 之后未至 t2 之前一致的数据应为 100+10-10+0=100

类似的，在 A 和 B 上也可以这样判断

CRDT 不是一个基于共识的系统，其使用一些简单的数学性质来确保事件的连续性。CRDT 快速将备份整合为和使用序列处理效果相同的通用的状态。由于 CRDT 不需要同步，一个升级会被快速地处理，并不被网络的延迟性所影响。一个最简单的 CRDT 的例子就是一个可复制的计数器，其能保证一致性是因为计数器增和减操作能相互通信（commute）。

CRDT 相比 OT 这类现在大多数实时文本协作系统应用的技术相比优势也很明显，其拥有很好的容错能力，以及很好的可伸缩性，同时也不需要像 OT 这样考虑不同的操作顺序，其具体实现机制也并不繁杂。因此，CRDT 应用场景也十分广泛，能应用到延时容忍网络中的计算，广域网中的延时容忍，断连操作，容错的 P2P 计算，数据聚集……

3.2 概念

3.2.1 原子和对象

一个进程可以存储原子和对象。一个原子是一个基本的不可被改变的数据类型，原子是由它的字面意思被识别；如果他们有相同的内容，那么原子可以被认为是相同的。原子的类型在这篇文章中被划分为整数，字符串，集合，元组，和他们的不可改变的操作。原子的类型可以被写成更低级的形式比如集合。

一个对象是可改变的，重复的数据类型。对象的种类是被大写化的。例如”Set”。一个对象拥有一个身份，可能是任意对象的一个内容，一个初始状态，并且一个由操作构成的界面。两个操作拥有相同的身份但是却定位在不同的进程中，这样的情况被称为另一个的复制品。

3.2.2 操作

该环境由未指定的客户端组成，这些客户端通过调用其接口中的操作来查询和修改对象状态，并针对他们选择的称为源副本的副本进行查询和修改。查询在本地执行，即完全在一个副本上执行。更新有两个阶段：第一，客户端在源处调用操作，该操作可能会执行一些初始处理。然后，更新将异步传输到所有副本。

3.3 CRDT 的一些类型

CRDT 有很多种应用场景不同的实现类型和方法，下面来简述一下 G-Counter 和 PN-Counter 两种实现。

3.3.1 G-Counter (Grow-only Counter

G-Counter 市一中基于状态的 CRDT，它为 n 个节点构成的集群维护了一个计数器，每一个节点被从 0 到 n-1 编号，通过 *myID()* 返回 ID。计数器的更新在后台进行，并且通过对 P 中所有元素取最大值完成。每个节点都维护在数组 P 中维护了自己的计数器，实现本地增加。更新函数通过比较函数的结果来单调递增地增加 CRDT 的内部状态。

3.4 文本实时协作中 CRDT 的实现

首先我们来举一个最简单的例子，假设有三个人 Alice、Bob、Oscar 同时对一个袋子（bag）进行操作。袋子支持增加物品和删除物品两种操作。如果刚开始袋子中只有一个苹果，此时 Alice 和 Bob 都想从袋子中拿走一个苹果，而 Oscar 却想往袋子中放一个苹果。这样就会产生问题了。显然，每个人对自己所保存的袋子的“状态”的基础上先执行自己要执行的操作，Alice 和 Bob 此时认为袋子中没有苹果，而 Oscar 的袋子中有两个苹果。此时 Oscar 收到了两个 Remove 操作，移除袋子中的两个苹果，袋子为空。然而 Alice 和 Bob 的状态不能确定。因为他们俩可能先收到 Remove 指令，再收到 Add 指令，此时袋子中有一个苹果。亦或先收到 Add 指令，再收到 Remove 指令，此时袋子中没有苹果。为了解决这个问题，我们要个每一个苹果一个独一无二的编号，这样，比如 Alice 和 Bob 同时想移除苹果 1，而 Oscar 想加入苹果 2，就不会造成问题了，最终三个人都会得到袋子中只有苹果 2 的状态。

对于文本的协同编辑，我们也采用同样的方法。我们给每一个原子（也就是每一个字符）赋予一个独一无二的位置标识符（PosID），满足一下三条原则

1. 每一个在缓冲区中的原子都有一个 ID

2. 任意两个不同的原子都有两个不同的 ID

3. 一个给定原子的 ID 在整个文档剩余的生命周期中保持不变

4. ID 之间有一个全序关系，和原子在缓冲区中的顺序一致

5. ID 取值空间是密集的，即：$\forall P, F : P < F \Rightarrow \exists N : P < N < F$

我们定义一个抽象的原子数据缓冲区的状态 T 是一个由 $(atom, PosID)$ 二元组构成的集合，状态 T 的内容就是由 T 中所有原子按照 $PosID$ 排列构成的序列。

每一个用户都会维持这个 CRDT 的一个副本，并且进行本地编辑

- $insert(PosID_n, \text{newatom})$
- $delete(PosID_n)$

这样，两个指向不同的 PosID 的操作是相互独立的。因为他们对于 CRDT 的作用效果是与他们的执行顺序无关的。那么现在只需要考虑并发的指向相同的 PosID 的两个操作。一个插入操作必定发生在一个删除操作之前签，所以他们不可能是并发的。最后，删除操作是幂等的（idempotent），因为删除掉了一个给定 PosID 的字符再次删除这个 PosID 的字符是无效的。因此，我们这样构建的一个缓冲区就构成了一个简单的 CRDT，实现了文本实时协作功能。

但是，在这里还有两个问题需要考虑。

- 两个客户端在同一时间生成了同样的 PosID（当他们并行地在同一个位置插入字符）
- 一个客户端生成了一个已经被使用过的 PosID（删除一个字符之后重新插入他们）

第二个问题很好解决，每个客户端自己只要维持一个记录，保证不会生成自己已经使用过的 PosID 即可。第一个问题同样可以用一个很简单的方法解决，我们只要将每一个 PosID 变成一个二元组即可，把 PosID 这个标识数字和产生该 ID 的客户端编号放在一起即可。这样就保证了两个客户端不会产生同样的 PosID。

4 技术依据

4.1 JavaScript 和 Node.js

4.1.1 JavaScript

JavaScript，是一门通过解释执行, 动态类型，面向对象（基于原型，可以用原型链的方式实现，每一个对象都有一个 prototype 属性，指定该对象的原型）的直译语言。它已经由 ECMA（欧洲电脑制造商协会）通过 ECMAScript 实现语言的标准化。它被世界上的绝大多数网站所使用，也被世界主流浏览器（Chrome、IE、Firefox、Safari、Opera）支持。它支持面向对象编程，命令式编程，以及函数式编程。它提供语法来操控文本、数组、日期以及正则表达式等，不支持 I/O，比如网络、存储和图形等，但这些都可以由它的宿主环境提供支持。

用 JavaScript 可以很方便的做到

- 嵌入动态文本于 HTML 页面
- 对浏览器事件作出响应
- 读写 HTML 元素
- 在数据被提交到服务器之前验证数据
- 检测访客的浏览器信息
- 控制 cookies，包括创建和修改等

不同于服务器端脚本语言，例如 PHP 与 ASP，JavaScript 主要被作为客户端脚本语言在用户的浏览器上运行，不需要服务器的支持。随著引擎如 V8 和框架如 Node.js 的发展，及其事件驱动及异步 IO 等特性，JavaScript 逐渐被用来编写伺服器端程式。且在近几年中，Node.js 的出世，让 JavaScript 也具有了一定的服务器功能，且在某些方面比 PHP 的效果更为显著。

4.1.2 Node.js

Node.js 是一个能够在服务器端运行 JavaScript 的开放原始码、跨平台 JavaScript 执行环境。Node.js 由 Node.js 基金会持有和维护，并与 Linux 基金会有合作关系。Node.js 采用 Google 开发的 V8 执行程式码，使用事件驱动、非阻塞和非同步输入输出模型等技术来提高效能，可优化应用程式的传输量和规模。这些技术通常用于资料密集的实时应用程式。

Node.js 大部分基本模组都用 JavaScript 语言编写。在 Node.js 出现之前，JavaScript 通常作为用户端程式设计语言使用，以 JavaScript 写出的程式常在用户的浏览器上执行。Node.js 的出现使 JavaScript 也能用于服务器端编程。Node.js 含有一系列内置模组，使得程式可以脱离 Apache HTTP Server 或 IIS，作为独立服务器执行。

Node.js 主要的作用就是相当于 Django 和 Flask 这样的 Python 后端，使得 JavaScript 可以脱离浏览器直接在服务器运行，让 JavaScript 可以直接编写服务器的后端，用 JavaScript 这一门语言就可以打通前后端。Node.js 也提供了报管理器 npm

- 允许用户从 NPM 服务器下载别人编写的第三方包到本地使用
- 允许用户从 NPM 服务器下载并安装别人编写的命令行程序到本地使用
- 允许用户将自己编写的包或命令行程序上传到 NPM 服务器供别人使用

用 npm 可以很方便的实现环境的配置，可以轻松在一台服务器上部署我们开发的应用

```
npm install
node app.js
```

4.2 Socket.IO

Socket.I 是一个面向实时 Web 应用的 JavaScript 库。它使得服务器和客户端之间实时双向的通信成为可能。它有两个部分：在浏览器中运行的客

户端库，和一个面向 Node.js 的服务端库。两者有着几乎一样的 API。像 Node.js 一样，它也是事件驱动的。

Socket.IO 主要使用 WebSocket 协议。但是如果需要的话，Socket.io 可以回退到几种其它方法，例如 Adobe Flash Sockets，JSONP 拉取，或是传统的 AJAX 拉取，并且在同时提供完全相同的接口。尽管它可以被用作 WebSocket 的包装库，它还是提供了许多其它功能，比如广播至多个套接字，存储与不同客户有关的数据，和异步 IO 操作。

Socket.IO 会自动选择合适双向通信协议，仅仅需要程序员对套接字的概念有所了解。

Socket.io 最大的特点就是封装 WebSocket 等协议实现了 TCP 客户端和服务器端全双工通信。其包含两个部分

- 一个 Node.js 服务器端
- 一个 JavaScript 客户端库

其具有极强的可靠性，即使在代理和负载平衡器以及防火墙和杀毒软件存在的情况下，仍然能提供稳定的连接。Socket.IO 在断连时也会尝试不断自动重连，直到服务器可用为止。同时 Socket.IO 也支持二进制序列化数据，任何序列化数据都可以发送，例如浏览器中的 ArrayBuffer 和 Blob 同时 Socket.IO 也提供了简单易于使用，用几条简单的命令就能实现事件的发送和事件的监听

```
io.on('connection', function(socket){
    socket.emit('request', /* */);
    // emit an event to the socket
    io.emit('broadcast', /* */);
    // emit an event to all connected sockets
    socket.on('reply', function(){ /* */ });
    // listen to the event
});
```

4.3 Codemirror

4.4 Showdown.js

5 创新点

参考文献

[1] Mehdi Ahmed-Nacer, Pascal Urso, Valter Balegas, and Nuno Preguiça. Merging OT and CRDT Algorithms. In *1st Workshop on Principles and Practice of Eventual Consistency (PaPEC)*, Amsterdam, Netherlands, April 2014.

[2] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, June 1989.

[3] Santosh Kumawat and Ajay Khunteta. Analysis of operational transformation algorithms. In Nitin Afzalpulkar, Vishnu Srivastava, Ghanshyam Singh, and Deepak Bhatnagar, editors, *Proceedings of the International Conference on Recent Cognizance in Wireless Communication & Image Processing*, pages 9–20, New Delhi, 2016. Springer India.

[4] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the jupiter collaboration system. In *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*, UIST '95, pages 111–120, New York, NY, USA, 1995. ACM.

[5] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. ACM.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[7] Chengzheng Sun and Rok Sosic. Optional locking integrated with operational transformation in distributed real-time group editors. In *In Proceedings of the 18th ACM Symposium on Principles of Distributed Computing*, pages 43–52, 1999.