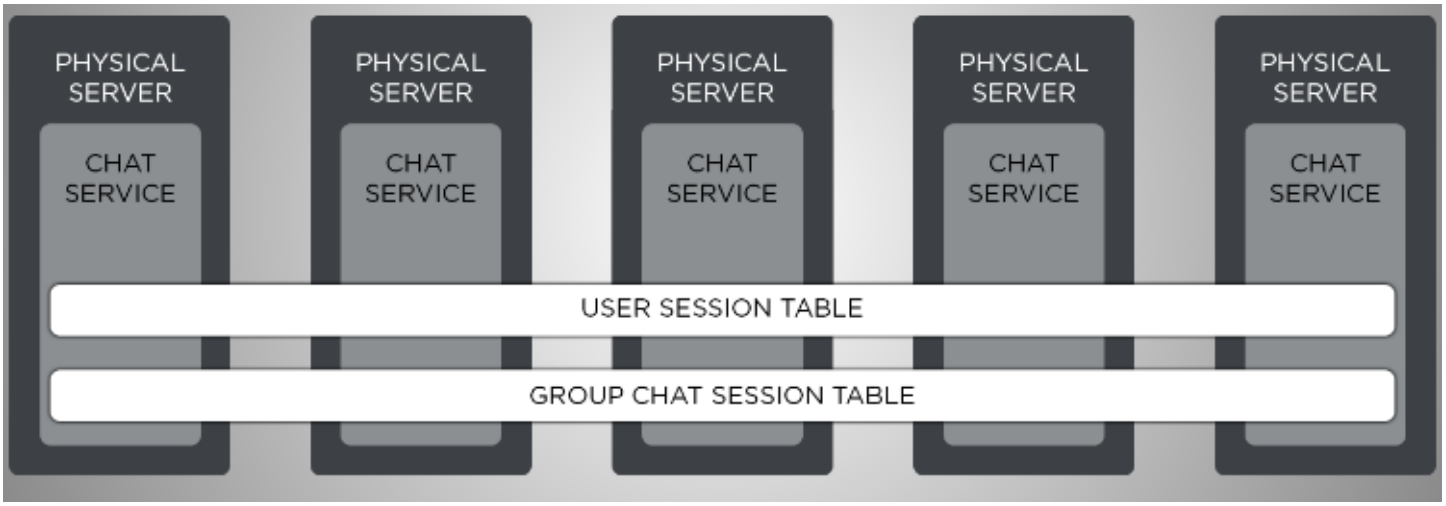
[译]拳头公司聊天服务架构：服务器篇

Michal Ptaszek · 2015-10-27 14:48

原文：Chat Service Architecture Servers (http://engineering.riotgames.com/news/chat-service-architecture-servers)

译者：杰微刊 (http://www.jointforce.com/jfperiodical/openhome)-张帆



英雄联盟玩家每天总计发送数百万的信息。他们邀请朋友双排，在英雄选择界面和队伍沟通英雄选择，在战局结束时向所有人敲出"GG"（Good Game）以表感谢。今年的7月21号（我随机挑选了一天），玩家在游戏中建立了170万新的好友关系——满满都是爱！每次玩家发送消息，都会触发一系列后台的操作，而正是这些后台技术驱动了Riot的聊天系统。

在之前的文章《聊天服务架构：协议篇 (http://engineering.riotgames.com/news/chat-service-architecture-protocol)》中，我论述了我们所选用的客户端服务器间通信协议：XMPP（Extensible Messaging and Presence Protocol， 可扩展通讯和表示协议）。今天我会适当地就服务端和基础设施架构的相关机制进行深入探讨，同时，我也会论述到目前为了保证服务器的可扩展性和健壮性，我们所做的努力和已经完成的工作。像上一篇文章一样，我希望任何需要为分布式客户端库添加聊天特性的开发者会对本篇文章感兴趣。

服务器硬件

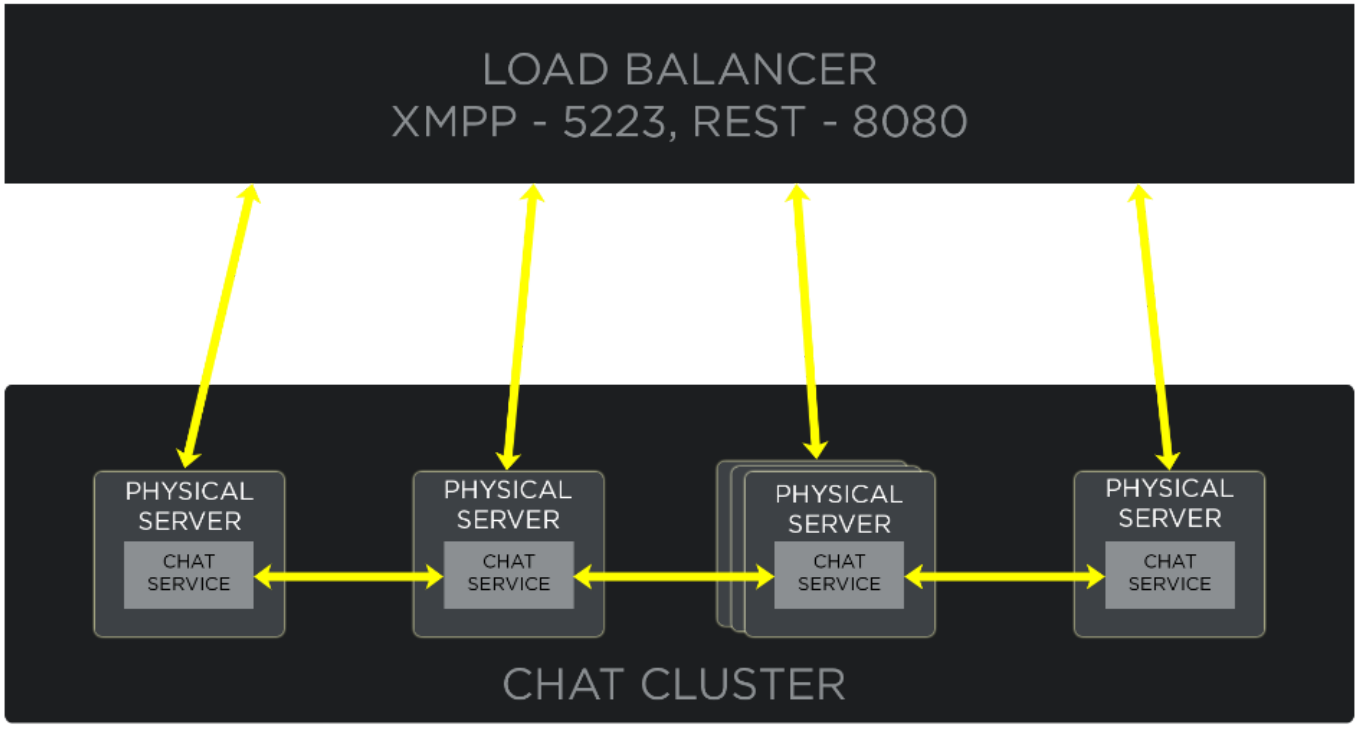
聊天系统的物理服务器应当具备这样的能力——确保服务对玩家来讲持续可用。这些服务器管理每个用户的聊天会话，同时能够保证必要的稳定性和应用安全验证隐私设置，如访问量流量限制、指标收集和日志记录等。

我们的聊天服务是按区域部署的（我们称每个区域为一个“分片”），这意味着每个英雄联盟区服拥有其自己的聊天集群，用于提供并且仅为该分片的玩家提供聊天功能。结果导致跨区服的玩家不能进行沟通交流，同时聊天服务器不能使用其他区域的数据。比如，北美服务器（NA）不能直接与西欧服务器（EU West）进行交流。

每个分片维护了一个由大量各种各样的物理服务器组成的聊天集群，这些服务器上运行着相同的服务器软件。每个区域的硬件规格参数都不尽相同，因为要考虑一系列的原因，如承载量需求、设备年限和硬件可用性——我们最新的服务器具备现代24核CPU、196GB的内存和固态硬盘（SSD），而较陈旧的服务器则使用24G内存和传统磁碟硬盘。

在聊天集群内，每个节点是完全独立和可替换的，这使得系统可以轻松的进行维护，从而提高了系统整体的容错性。一个集群中的节点数约为6~12台机器。尽管我们可以在每个区域运行更少的机器数量，但是为了给容错提供足够的空间和适应将来的增长需求，还是保证了足够的数量。如果月到升级需要关闭服务器，举例来讲，我们可以关闭单个集群中一半的节点数来确保不会中断为玩家提供的服务。

下面是一个简单的图表，展示了我们聊天服务的集群架构



在过去一段时间，我们遇到过大量由于硬件或者网络故障引起的意外事件，为了解决这类问题，我们不得不将个人服务器关闭几天，然而正是因为系统具备较好的容错机制，对玩家来讲，服务持续进行并没有中断。此外，我们还在客户端实现了这样一个逻辑——帮助已连接到关闭服务器的玩家自动重新获取他们的链接。

具体实现

拳头的聊天服务器主要是用Erlang（如果你对该语言感兴趣，可以点击查看这个视频https://www.youtube.com/watch?v=xrIjfIjssLE）写的，此外我们还用C来实现一些下层操作，如XML解析、S

SL握手、字符串操作等。我们聊天服务的代码库，大约10%是C语言实现，90%是纯Erlang（忽略外部库和Erlang VM本身）。

在用C语言开发服务器组件之前，我们花了大量的时间分析和优化现有的Erlang代码库。我们尝试在使用全部现有工具的前提下，找出不同抽象等级下潜在的并发和效率瓶颈：

① 对于简单的调用计数，我们使用cprof (http://www.erlang.org/doc/man/cprof.html)，cprof框架极其简单，却能够具有验证我们是否在正确的时间进行了正确调用的能力。

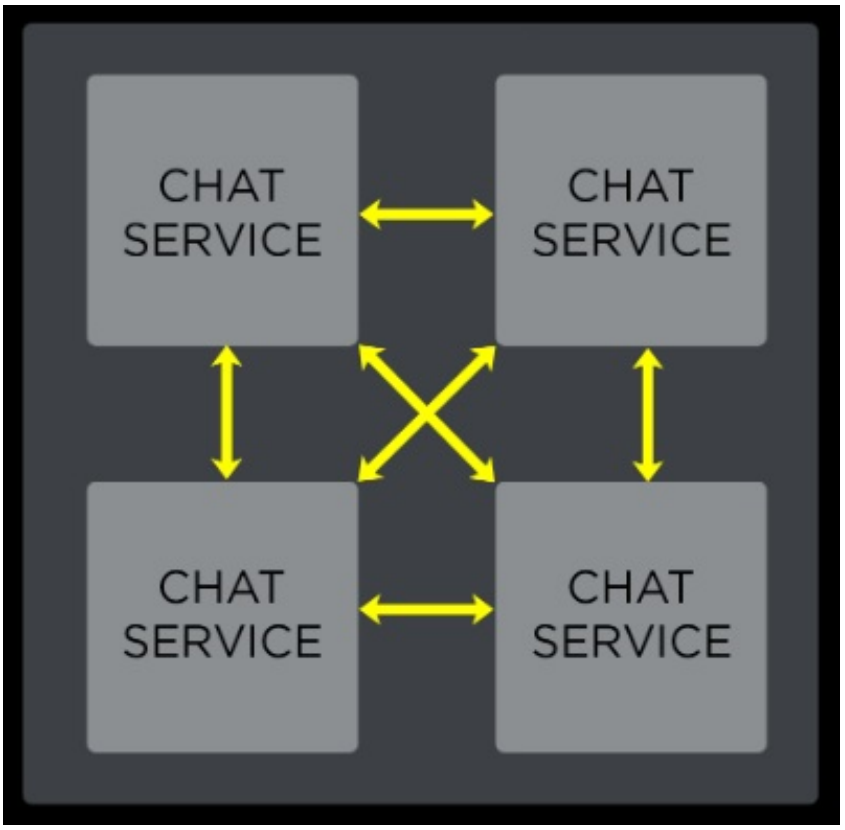
② 为了进行更详细的分析，我们运行了fprof (http://www.erlang.org/doc/man/fprof.html)。不幸的是，fprof对测试服务器会产生较大的影响，所以不能在一个完整的负载测试期间使用——然而，它却能够使我们更详细的了解到如何和何时进行调用。其中包括单个函数在它执行期间花费的时间（own time，独占时间），包含调用函数所花费时间在内的全部时间（accumulated time，累计时间）和过程结果的分组。所以这些帮助我们找出系统执行过程中的CPU密集区域。

③ 当检测并发瓶颈时，percept (http://www.erlang.org/doc/man/percept.html)和lcnt (http://www.erlang.org/doc/man/lcnt.html)十分友好。这些工具帮我们尽可能的识别所有并行化数据处理的时机，以便是我们可以利用所有的可用核心。

④ 在操作系统层面，我们采用了传统分析工具，比如mpstat (http://linux.die.net/man/1/mpstat),vmstat (http://linux.die.net/man/8/vmstat),iostat (http://linux.die.net/man/1/iostat),perf (http://linux.die.net/man/1/perf)，还有一些/proc (http://linux.die.net/man/5/proc)的文件系统文件。

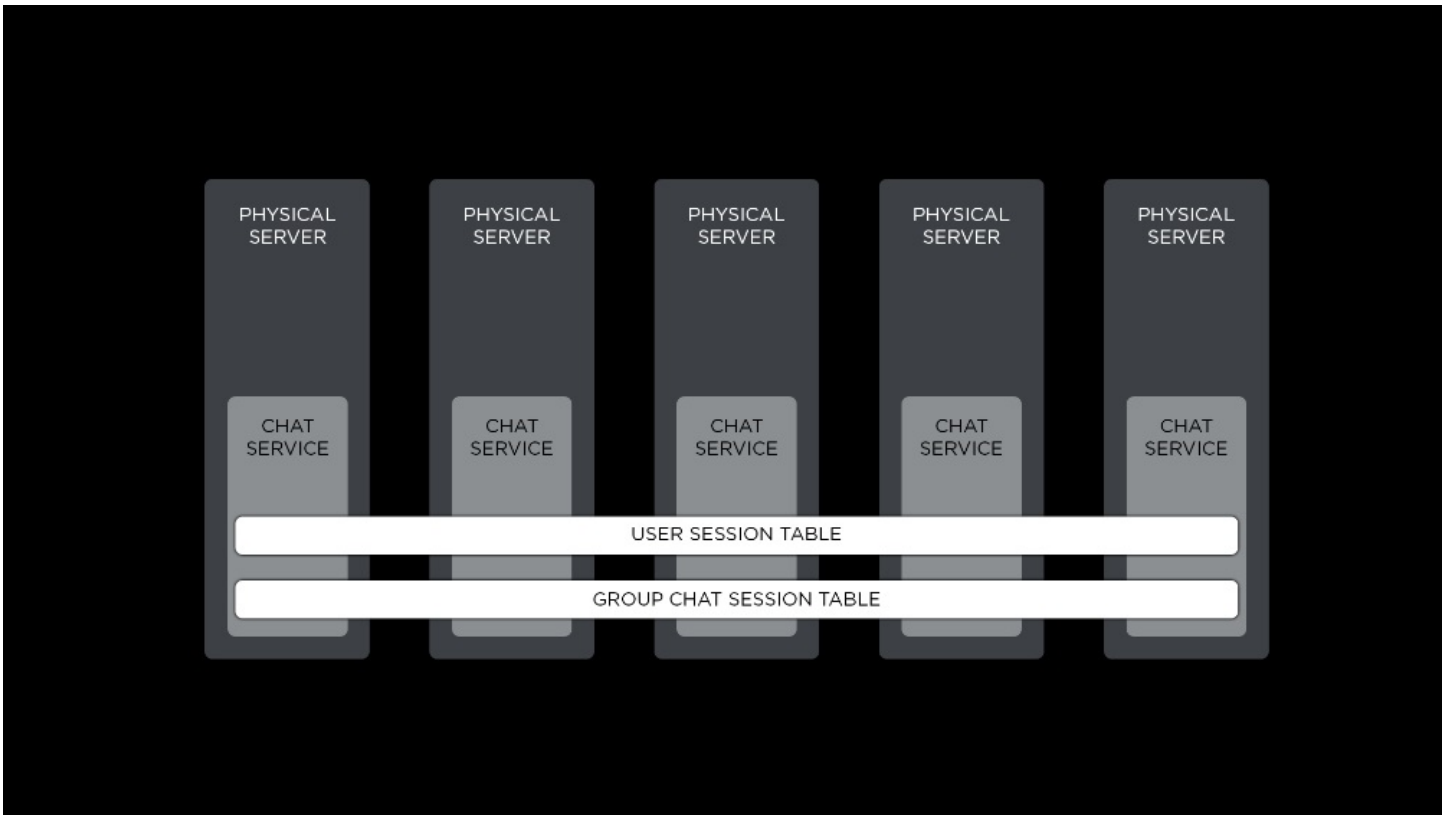
我们发现，大部分珍贵的CPU执行周期和内存分配发生在处理文本和数据流时。由于Erlang的terms (http://www.erlang.org/doc/reference_manual/data_types.html)被定义为不可修改，不管我们如何优化代码，执行任何字符串密集的排序操作将十分“昂贵”。所以，我们使用C来重写最笨重的字符串操作的系统部件，同时，使用HiPE (http://www.erlang.org/doc/man/HiPE_app.html)来编译部分标准库——这使得CPU占有率性能提高了60%以上。此外，我们发现，对每个玩家的会话处理，内存分配由150kb降低至25kb。

我们尝试遵从“Erlang/OTP设计原则和最佳实践”（best practices and Erlang/OTP design principles）(http://www.erlang.org/doc/design_principles/users_guide.html)，所以拳头的聊天服务器形成了一个全关联的集群，即集群中的每两个聊天服务器之间由Erlang VM维护单独的一个持久TCP连接。服务器使用这些连接来传达Erlang分布式协议，进行内部交流。至此，无论如何详尽的测试，我们再次交流模型上再也找不到任何瓶颈。



前面我提到过，集群中的每个服务器都是一个完全独立的单元——这使得每个服务器在任何时间都可以独立操作，独立运行，独立提供服务。为了提供充分的可扩展性/伸缩性，我们将系统设计成节点间共享尽可能少的数据。因此，聊天集群中的服务器仅共享一小部分内部键值表，这部分表完全复制在内存中，通过一个高度优化的Erlang分布式数据库管理系统进行通讯，这个系统被称作Mnesia（如果你想学习更多有关Mnesia的内容，我强烈建议你去看一下erlang.org (http://www.erlang.org/doc/apps/mnesia/Mnesia_chap1.html)上的材料和这个网站learnyousomeerlang.com (http://learnyousomeerlang.com/mnesia)）。这些表将玩家或者组聊天识别符（在XMPP中被称作JIDs）与Erlang处理器进程进行映射，后者维护了一些上下文特定的数据。

对单独的玩家聊天，这些数据中可能会包括链接套接字（connection socket）、好友列表和限速器；对于群组对话、这些数据则包括房间名册和聊天记录。每次在玩家或者群组对话间有需要路由任何类型的信息时，服务器会问询这些键值表以通过JIDs来处理他们的会话句柄。因为每个服务器持有一份session表的完全拷贝，所以所有的都操作都是在本地发起，减少了整体的路由延迟。

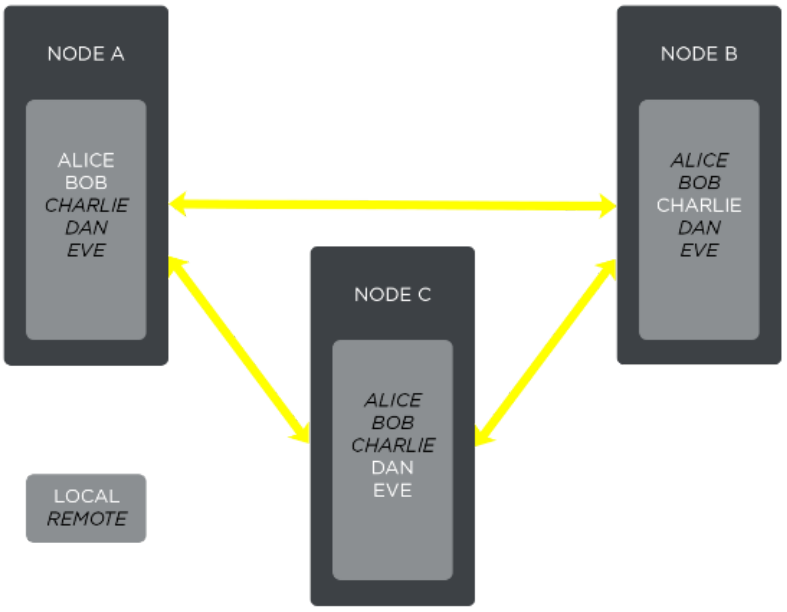


上述做法的负面影响就是每个应用于这些表的更新必须被复制到其他所有已连接的聊天服务器中。据我们所知，这些表是唯一可能会阻止我们顺利线性扩展聊天集群的因素——然而，负载测试显示我们可以将聊天集群增加至30台服务器而没有任何性能损耗。

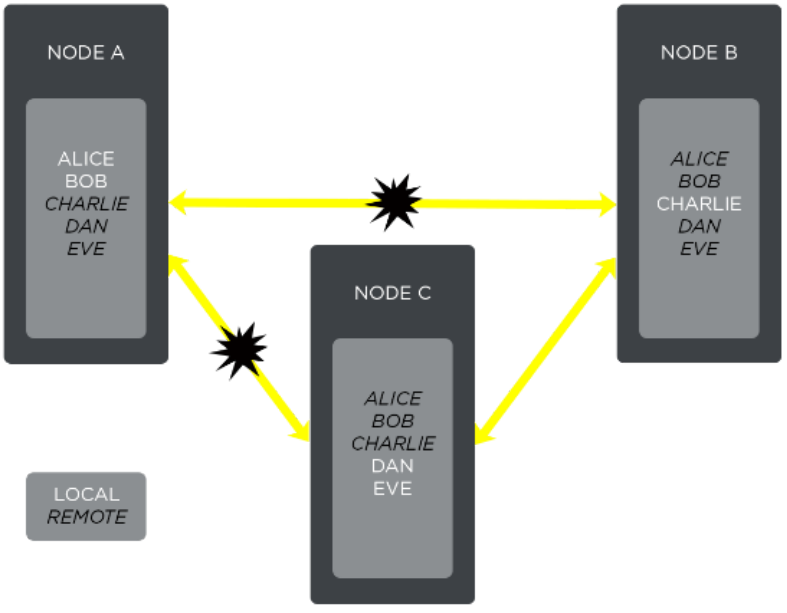
为了保证玩家的会话数据或者群组聊天数据在网络切割或者服务器故障时保持同步，每个聊天服务器运行订阅了关于集群拓扑变化的VM内部通 (http://erlang.org/doc/man/net_kernel.html#monitor_nodes-1)知Erlang进程。当任何事件产生，比如聊天服务器宕机，其他节点将从它们的本地表中移除脱机节点上正在运行的所有会话条目。一旦节点恢复上线（通常是在重启或等待网络连接恢复之后），该节点会向其他集群成员推送自己的本地状态，并且下载其他节点的状态。通过这种模式，聊天服务器对其自己的数据而言扮演了绝对权威的角色，对其他聊天服务器的数据，完全信任。

为了解释这个设计，让我们一起考虑下面的示例：

① 我们有一个三台聊天服务器组成的集群：节点A（玩家Alice和Bobby连接到该节点）、节点B（玩家Charlie连接到该节点）和节点C（玩家Dan和Eve连接到该节点）。所有的玩家可以立刻进行交流:

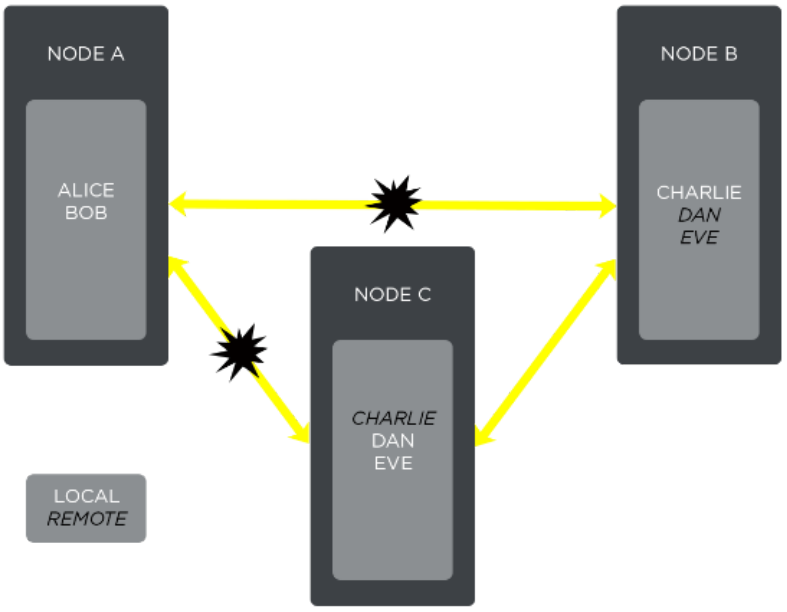


② 网络故障导致节点A与集群中的其他节点脱离，下次其他机器尝试与A进行通讯时，会遇到一个关闭的TCP套接字并产生一个由信号群集拓扑改变引发的事件。这是网络断裂的一个十分典型的案例。在这个案例中，Alice和Bob就像被隔离到单独的岛屿上，不能与Charlie、Dan和Eve进行通讯，然而Charlie、Dan和Eve之间仍可正常通讯:

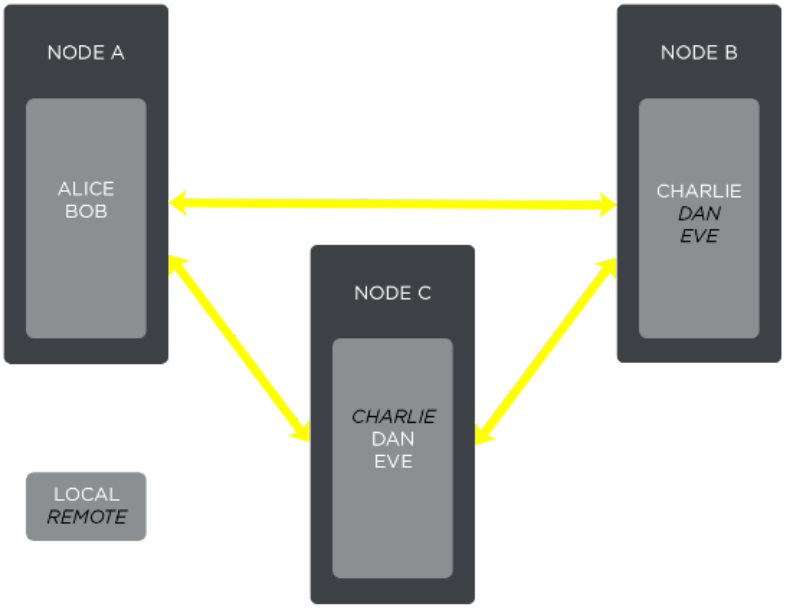


③ 现在，节点B和C不能访问单独一侧的节点A，当Erlang VM将集群拓扑变更事件传递给订阅处理程序，节点B和C会舍弃节点A上的玩家会话引用。尽管Alice和Bob仍可访问服务并相互之间可以聊

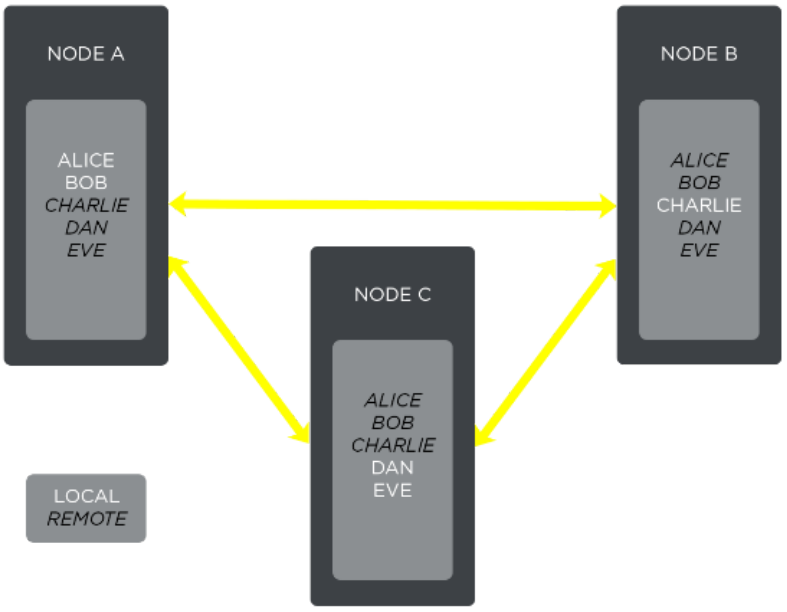
天，但是他们不能与Charlie、Dan和Eve进行沟通。【译者注：原文中的下图中所示，节点A其实也把Charlie，Dan和Eve的会话引用舍弃掉了，文中并没有指明】



④ 网络连接恢复后，节点A重新与节点B和C之间建立TCP链接。不幸的是，连接建立后，节点A还不知道节点B和C，所以Alice和Bob仍然不能与Charlie、Dan和Eve沟通:



⑤ 最后，Erlang VM再次发起了集群拓扑变更事件，这次发出的通知是服务器加入。节点A下载节点B和节点C的会话数据到A本地Mnesia表，节点B和C同理将节点A的会话数据下载到本地。现在集群再一次复制完毕，可以允许Aclice、Bob、Charlie、Dan和Eve之间所有的通讯:



这个设计可以使我们构建健壮可自修复的基础架构，不再需要在连接失败的情况下立即手动干预。所以，服务可以自动从网络异常中恢复，并对玩家恢复其功能。

数据流

基于上面谈到的服务器实现，让我们来审查下提供给玩家聊天的软件过程。无论任何时候，当玩家客户端链接到聊天服务器时，客户端便向其公共的XMPP终端开放了一个持久的、加密的（采用AES256-SHA加密算法）TCP链接。集群负载均衡器会在后端选择聊天服务器中的一个，并且将该玩家的会话指定分配到这个服务器。目前，我们使用一个正则循环负载平衡策略以在所有可用的服务器之间分发负载均匀。

一旦链接建立，聊天服务器会创建一个新的专用的Erlang进程来处理该玩家的会话。这个进程（又被称作c2s，即client to server, 客户端到服务端）维护了玩家的TCP套接字、XML解析器 (http://expat.sourceforge.net/)实例、好友列表和黑名单、最后状态数据、召唤师名称、速率限制配置和其他重要的服务器所使用的用于提供玩家体验的详细信息。在链接中，我们的系统会立即要求身份验证，校验该玩家身份。在这里，我们试用了兼容第三方客户端的XMPP标准身份验证机制 (http://xmpp.org/extensions/xep-0034.html)。

为了进一步说明，我们还是举Alice和Bob的例子。二位都是青铜2的玩家，期待有一天可以打上王者段位。为此，Alice和Bob天天一起训练，在召唤师峡谷磨练他们的技术。

Alice想给Bob发信息，邀请Bob一起双排去打晋级青铜1的晋级赛。下面是一旦Alice通过验证链接到聊天服务器后台的一系列操作：

① Alice的游戏客户端通过其c2s进程维护的加密的TCP链接发送了一条XMPP信息（包含这样一条信息“要来一起打晋级赛生死局吗”）到聊天服务器

② Alice的c2s进程收到信息后进行解密喝转换成XML

③ 转换之后，该进程在这条消息上进行了几个验证，包括速率限制遵从性、防欺骗验证、黑名单和好友列表成员测试

④ 通过验证之后，Alice的c2s在其内部的映射表中查找Bob的会话句柄，并验证是否可达

⑤ 如果Bob当前不在线，则信息会被持久化到数据存储中，下次Bob登陆游戏之后再次发送他。根据分片的不同，数据库选用MySQL（对于老式环境）或者Riak（对于新的分片）。数据存储将是我下一片讨论聊天服务架构文章的主题

⑥ 如果Bob当前在线，则服务器将使用标准Erlang信息传递机制 (http://learnyousomeerlang.com/the-hitchhikers-guide-to-concurrency)将信息发送至他的c2s进程

⑦ 当Bob的c2s进程接收到信息，将会像前面那样进行一系列校验

⑧ 校验结束，进程将将消息序列化为XML，然后发送至Bob的TCP套接字

⑨ 最终，Bob的游戏客户端可以接收到信息，并展示在合适的信息窗口

显而易见，一旦收到信息，Bob会很乐意加入Alice，两人将carry他们的团队，很快就可以晋级青铜1了。

内部接口

除了直接提供玩家聊天特性，Riot的聊天服务器还公开了几个私有的内部消费相关的REST接口。一贯地，服务使用这些REST端点来访问社交图谱（由玩家之间的好友关系确定）

例如：

1) 对于内容馈赠，如英雄皮肤，在游戏内部的商店背后，由服务器验证他们的好友关系是否足够“成熟”——这避免了威胁账号的恶意赠送

2) 建立战队的功能使用了LoL（League of Legends简写）的社交图谱，用来构建一个建议好友列表供玩家在建立战队时选择邀请的好友

3) 对于联盟相关的服务，系统会在后端查询好友列表用来决定哪些是新手玩家，偏向于将这些玩家与其好友放置在一方——即匹配时，更有可能与自己的好友在一个联盟中。

这里拿一个真实的例子，考虑一下请求ID13131召唤师的好友列表：

```
GET /friends/summoner/13131 HTTP/1.1 … < HTTP/1.1 200 OK < server: Cowboy < connection: keep-alive [ { "ask": "none", "askmessage": "", "createdat": "2015-06-30 10:52:26", "group": "Work", "nick": "Riot Teemo", "note": "top laner!", "subscription": "both", "summonerid": 112233 }, { "ask": "none", "askmessage": "", "createdat": "2015-06-25 11:25:07", "group": "Family", "nick": "P
```

entakill Morg", "note": "Mom", "subscription": "both", "summonerid": 223344 }, { "ask": "none", "askmessage": "", "createdat": "2015-06-17 17:57:17", "group": "Work", "nick": "Jax Jax Jax", "note": "plays only Jax?", "subscription": "both", "summoner_id": 334455 }, ⋯] ```

为了支持这些请求，聊天服务器运行了Cowboy (https://github.com/ninenines/cowboy),一个用于处理其他内部服务传来的HTTP请求的嵌入式Web服务器。为了方便集成，每个后端通过Swagger (http://swagger.io/)返回JSON对象来填充请求。

尽管当前，我们使用了一个集中的内部均衡器来分发这些请求，在将来，我们更倾向于使用自动发现模型 (https://github.com/Netflix/eureka)和客户端的负载均衡机制。服务可进行自配置可以使我们在不用进行网络的重新配置的前提下便可以动态的重新部署集群大小，同时还可以十分简单的建立新的分区（对内提供测试或者对外提供玩家使用）。

结语

在单独的一天之中，Riot的聊天服务器通常会路由转发大约十亿的事件（如在线状态、信息等），并且处理上百万的REST查询。尽管现在系统并不完美，基础设施的健壮性和可扩展性还是能够保持聊天服务对玩家持续可用——今年，系统由于时不时出现的硬件问题常需要关闭某一个独立的服务，但是自修复的特性为Riot的所有区服提供了5/9的运行时间(five 9's of uptime这么翻译对吗？）。尽管我在这里仅仅探讨了很表层的内容，但是我还是希望能对于如何在服务器端增加了聊天功能进行一个抛砖引玉。

如有任何问题或者评论，欢迎留言，我很期待大家的留言。这个系列的下一篇文章，也是最后一篇文章，将会着重关注Riot聊天服务的数据库部分，届时见 ~