基于 Actor 模型的高性能分布式 XMPP 服务器①

陈 昊[1,2], 高楚舒[1], 魏 峻[1], 叶 丹[1]

[1](中国科学院软件研究所软件工程技术研究开发中心, 北京 100190)
[2](中国科学院大学, 北京 100190)

摘 要: 云计算和移动互联网的高速发展, 使得云端服务器需要同时和大规模客户端保持实时交互, XMPP 消息通信技术使用基于 TCP 长连接的方式来实现这一功能. 然而, 现有的 XMPP 服务器系统大多基于传统的并发模型设计, 整体性能较差, 无法应对大规模并发的需求. 本文针对 XMPP 服务器的特点, 提出了一种基于 Actor 模型的 XMPP 服务器架构设计, 并给出了一种基于一致性哈希的分布式消息路由算法, 有效提升了系统的并发度、弹性扩展能力, 以及消息传递的效率. 实验表明基于本文方法实现的系统相比于现有其他系统, 性能有很大提升, 可以适应大规模并发的场景.

关键词: Actor 模型; XMPP; 分布式系统; 即时通讯

High-performance Distributed XMPP Server Based on Actor Model

CHEN Hao[1,2], GAO Chu-Shu[1], WEI Jun[1], YE Dan[1]

[1](Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)
[2](University of Chinese Academy of Sciences, Beijing 100190, China)

Abstract: The cloud servers need to interact with a large number of clients simultaneously in real time in the cloud computing and the mobile internet environment. Real time communication between servers and clients is usually implemented with XMPP and TCP long connection. However, current XMPP servers, which are usually designed in the traditional concurrent model, can hardly deal with such large number of concurrent connections due to their limited performance. In this paper, we propose a new XMPP system design with the Actor model, and develop a decentralized message routing algorithm based on consistent hashing, to achieve better concurrency, scalability and efficiency. The experiment results show that our system overperforms a popular open source XMPP server in the scenarios of high concurrency.

Key words: Actor model; XMPP; distributed system; instant messaging

近年来, 随着云计算和移动互联网的高速发展, 二者的结合也越来越紧密, 二者通常被合称为“移动云计算”. 由于受到计算能力、存储能力、电量消耗等因素限制, 移动设备在面对复杂数据处理时往往无能为力, 需要云计算强大计算能力的支持. 因此, 目前移动云计算的发展趋势正是将客户端轻量化, 把复杂的数据处理移至云上, 并加强“云”和“端”二者之间的联系[1].

实现云端服务器和客户端实时通信的一种常用方法是在服务器和客户端之间保持 TCP 长连接, 并使用 XMPP 协议进行数据交换. 这种软件架构不仅可以很好地满足移动终端获取信息的实时性需求, 同时也避免了传统的基于 HTTP 定期轮询的方式对移动智能终端的网络流量、电池、CPU 等资源的大量消耗. 因此, XMPP 协议在移动互联网领域得到了广泛的应用, 例如即时通讯[2]、通知推送[3]等.

XMPP(Extensible Messaging and Presence Protocol, 可扩展通讯和表示协议)是一种以 XML 为基础的开

① 基金项目:国家自然科学基金(61173005);国家科技支撑计划(2013BAH05F03)
收稿时间:2015-01-22 收到修改稿时间:2015-03-09

放式实时通信协议[4]. XMPP 并不是一项新事物, 它的前身 Jabber 项目最初由 Jeremie Miller 于 1998 年提出, 起初只是为了设计一套开源、统一的即时通讯协议, 使不同的即时通讯客户端能够互联互通[5]. 由于 XMPP 基于 XML 语言实现, 它继承了 XML 语言超强的灵活性和扩展性. 随着 XMPP 的逐渐发展成熟, 其涵盖了消息传递、连接加密、实体验证等一套完整的实时通信机制.

但是, 现有的 XMPP 服务器大多具有以下几个缺点: 单机支持并发有限、集群方案不成熟、性能差. 对于普通开发者来说, 想要基于现有系统构建一套面向大规模高并发场景的即时通讯服务器或通知推送服务器具有很大难度.

因此, 本文的主要研究内容是设计并实现一个高性能、支持大规模并发、易于扩展的分布式 XMPP 服务器.

1 问题分析

XMPP 服务器的核心功能是消息转发, 即发送方客户端将消息通过 TCP 长连接发送至服务器, 服务器再将其转发至接收方客户端的过程.

对于一个面向大规模云应用的 XMPP 服务器来说, 服务端需要同时和数百万、数千万客户端保持长连接, 每秒钟需要并发处理数以万计的消息转发任务. 而消息应用的特点是对延迟时间比较敏感, 因此要求 XMPP 服务器具有很强的并发处理能力. 同时, 由于内存、CPU、网络带宽等因素的限制, 单台服务器仅能支持数万至数十万的并发连接. 所以要求 XMPP 服务器支持集群化, 并且具备良好的横向扩展能力, 以满足大规模并发连接的需求.

然而, 现有的 XMPP 服务器, 例如 Openfire[6]、Vysper[7], 大多采用了传统的并发模型. 我们知道, 使用传统并发模型构建高并发、高容错、可扩展的分布式程序是十分困难的. 因为在传统并发模型中, 线程是最小的并发粒度, 当不同线程访问共享内存时, 必须使用锁、信号量等同步机制以保证程序的正确性, 但是如果对其处理不当又有可能造成死锁、活锁、线程饥饿等问题. 同时, 系统组件之间一般通过同步方法调用的方式进行交互, 是一种紧耦合的关系, 可能因为一个环节的等待而阻塞整个流程. 因此, 传统并发模型很难适用于 XMPP 服务器这样的大规模高并发场景.

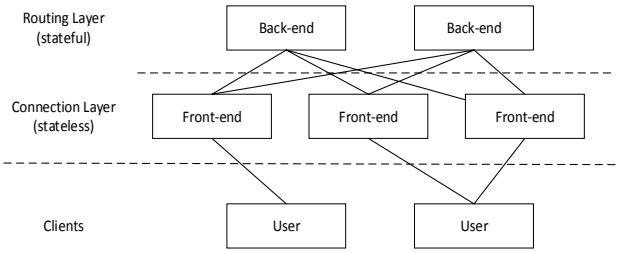

图 1 XMPP 服务端两层架构

另一方面, 为了提高分布式系统的横向扩展能力, 要尽量把节点设计为无状态的. 如图 1 所示, 分布式 XMPP 服务器通常采用两层架构, 也就是把系统分为无状态的前端节点和有状态的后端节点. 前端也叫做连接层, 只负责处理 TCP 连接, 客户端可以选择任意一个服务器节点建立连接. 因为是无状态的, 所以连接层节点可以任意扩展. 后端也就是消息路由节点, 维护用户到节点的映射关系, 负责将用户消息转发到用户所连接的节点, 是系统中唯一有状态的部分. 因此, 路由模块的效率、扩展能力、容错性、可用性是影响分布式 XMPP 系统性能的关键因素.

而现有的开源 XMPP 系统的另一个主要缺陷体现在路由模块的性能上. Tigase[8]的解决方案是不保存路由信息, 而是将消息转发至所有节点. 这种方案的明显缺点在于: ①集群没有起到分散负载的作用, 仅仅提高了可用性; ②造成了不必要的网络传输. 因此 Tigase 只适用于小规模的集群. 而 Ejabberd[9]的解决方案是使用一个外部的分布式内存数据库保存映射关系, 其缺点在于每次发送消息之前都要从数据库中查询路由信息, 效率不高. 还有一种常见的解决方案是中心式 Router, 即选择一个节点作为 Master, 维护路由信息表, 所有消息都经由该节点转发, 此方案的主要缺点在于 Master 节点会成为单点瓶颈.

综上, 现有 XMPP 服务器性能较差的两个主要原因是: ①传统并发模型限制了系统的并发能力和扩展能力, ②分布式消息路由算法效率低下. 因此, 本文将主要从提高 XMPP 系统的并发能力和扩展能力、设计高效的分布式消息路由算法两方面来介绍如何设计并实现高性能的分布式 XMPP 服务器.

2 基于Actor模型实现高并发、易于扩展的XMPP系统

针对基于线程的传统并发模型限制 XMPP 系统并

发能力和扩展能力的问题, 本文的解决思路是采用 Actor 模型替代传统模型来设计并实现 XMPP 服务器. Actor 模型[10]是一种并行计算的编程模型, 在 Actor 模型中, Actor 是系统的最小单元, 一个系统可以由千千万万个轻量级的 Actor 组成. Actor 是封装了状态和行为的计算实体, 不同 Actor 之间完全隔离, 不共享数据. 每个 Actor 都有一个信箱, Actor 之间只能通过发送异步消息的方式进行通信. 因此, 也可以把 Actor 看作一个应用层面的轻量级进程. Actor 反复不断地处理收到的消息, 并做出响应: 发送消息给其他 Actor, 创建新的 Actor, 或者改变自己内部的状态和行为[11].

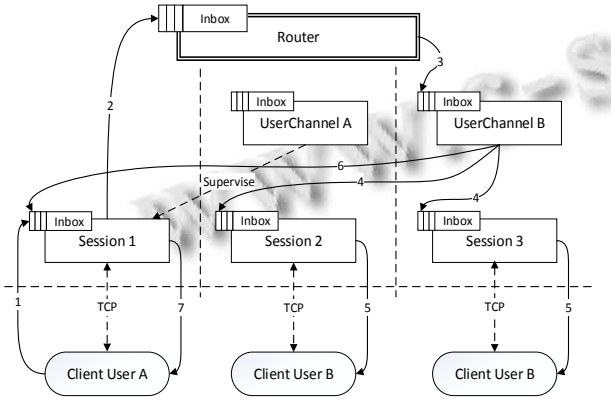

图 2　基于 Actor 模型实现消息传递处理流程

图 2 展示了如何采用 Actor 模型来实现消息传递的过程. 图中的每个带有 Inbox 的矩形表示一个 Actor 对象: Session Actor 用于维护 TCP 连接上的会话消息, 并负责和客户端进行 IO 通信; UserChannel Actor 是一个用户接收和处理消息的通道, 所有发给该用户的消息都会由相应的 UserChannel 处理后发到对应的 Session, 同时它也负责监控该用户当前登录的所有 Session 的状态; Router 是全局唯一的一个(或一组)Actor, 负责将消息路由给相应的 UserChannel(Router 的设计是决定分布式消息系统性能的关键因素, 本文将在下一节详细介绍 Router 的设计和实现). 如图所示, 客户端首先可以选择任意一个服务器节点建立 TCP 连接, 连接并登录成功后, UserChannel 开始监控这个 Session. 当用户 A 给用户 B 发消息时, 消息首先由 Session 1 发送至 Router, Router 将其路由至 UserChannel B 所在的节点, 最后再由 UserChannel B 把消息发送给用户 B 的所有活跃 Session(2 和 3), 并向 Session 1 发送一个异步的成功反馈.

从图中可以看出, 相比于传统并发模型, 使用 Actor 模型实现 XMPP 服务器的好处在于: 首先, 由于所有消息发送都是异步的, 对于 Session Actor 来说, 把消息放到 Router 的收件箱之后就可以马上处理下一个消息, 而不需要等待整个消息传递过程结束, 只要在消息送达之后把反馈作为另一个异步消息发回即可(曲线 6), 因此提高了并发处理消息的能力, 并缩短了响应时间. 其次, Actor 模型的核心思想是把一个复杂任务分解成由多个 Actor 承担的独立的子任务, 以及他们之间消息传递任务, 然后再由一个统一的调度器来调度这些任务的执行, 所以可以充分地利用多核资源, 纵向扩展系统. 最后, 因为 Actor 之间的消息传递是松耦合的, 所以无论 Router 部署在本地还是远程节点, 都不影响其他 Actor 的实现, 所以只需要简单修改 Router 的实现方式, 就可以很容易地将单机的 XMPP 服务器变成分布式系统, 即横向扩展系统.

而基于 Actor 模型设计系统的最主要难点在于, 它要求任务必须是可以细分成独立子任务的. 对于 XMPP 服务器来说, 一次消息处理过程通常需要经过多个系统组件, 每个组件的功能都是简单地处理消息体并传递给下一个组件, 因此组件之间是松耦合的关系, 并且整个过程中间没有事务性要求. 因此, 我们可以很容易地将整个消息处理过程细分为一系列独立的子任务, 从而利用 Actor 模型的优势, 设计适用于高并发通信服务场景的 XMPP 服务器.

3　基于一致性哈希的高效消息路由算法

针对现有 XMPP 系统路由功能效率较低的问题, 本文设计了一种基于一致性哈希[12]的去中心式(decentralized)路由算法. 如图 3 所示, 算法将哈希值的空间表示为一个大小为 2^{32} 的环, 然后对于每一个服务器节点, 以 *hash*(*ip* + *port*)的方式计算其哈希值, 并将其映射到哈希空间上. 之后对于某一个消息实体 m, 首先通过公式 *hash*(*receiverId*)计算出它在哈希环上的位置, 然后再沿环顺时针找到下一个节点, 并将 m 发送至该节点上的一个 LocalRouter Actor, 最后再发送至 UserChannel. LocalRouter 在这里除了充当一个本地的路由 Actor, 同时也负责 UserChannel 的生命周期控制. 同时, 为保证节点在哈希环上分布均匀, 算法将物理节点映射为环上随机分布的多个虚拟节点. 这样,

当节点 D 出现故障而宕机时, 只需将 A2 和 D1 之间的 UserChannel 移至 B2 节点, A1 和 D2 之间的 UserChannel 移至 C2 节点, 而不影响系统中的其他 UserChannel.

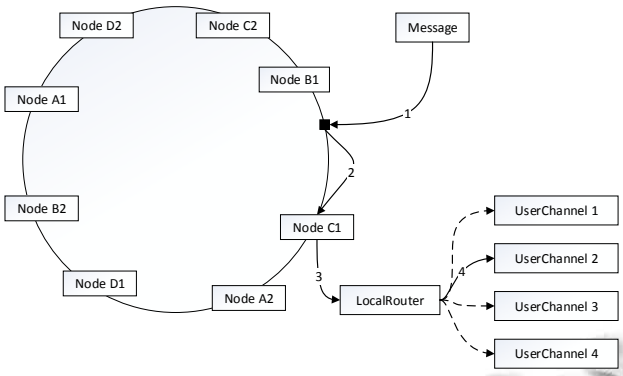

图 3 基于一致性哈希的去中心式 Router

这种方案相比于之前描述的几种方案而言, 主要的优点在于: ①系统中没有单点瓶颈和单点故障; ②可以根据哈希值直接算出目标节点, 不需要查询数据库, 效率高; ③节点之间需要同步的数据少, 只有节点位置信息, 网络开销低; ④可以动态地添加或删除节点, 而不影响其他节点的运行. 但是仍有一点不足之处, 由于 Actor 只能顺序地处理接收到的消息, 所以当消息数据并发量大时, 本地的 LocalRouter 仍可能成为一个性能瓶颈.

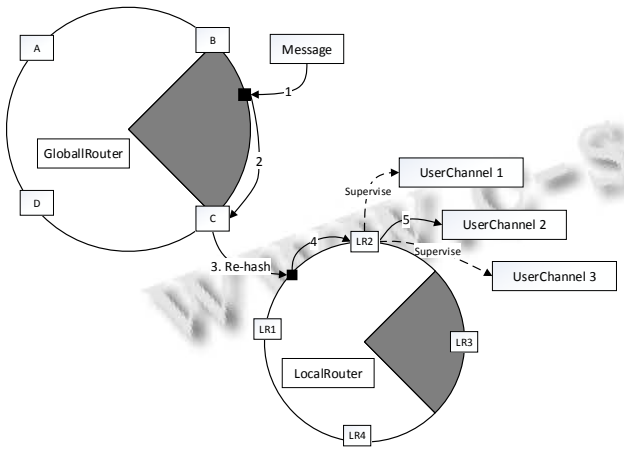

图 4 改进的二次哈希算法

因此, 本文在此基础上又提出了一种改进的二次哈希算法. 如图 4 所示, 算法的主要思想是将 LocalRouter 也转化成一致性哈希环, 由多个 LocalRouter 子节点分别管理一部分 UserChannel, 在环上寻找子节点的算法和 GlobalRouter 相同. 由于该哈希环的所有节点都在本地, 所以没有额外的网络传输开销, 并且可以很容易地根据负载情况动态地调节子节点的个数. 这种方法不但解决了 LocalRouter 的单点瓶颈问题, 同时也避免了因为 GlobalRouter 上分布节点太多而造成的同步节点数据时网络开销大的问题. 最后, 值得注意的时, 在 GlobalRouter 和 LocalRouter 之间需要重新计算消息体的哈希值, 否则将会出现哈希值偏移的问题, 即哈希值在 LocalRouter 上的分布范围仅限于 GlobalRouter 上两个节点中间的部分(图 4 中的两个扇形阴影部分), 而使得 LocalRouter 哈希环上的负载分布不均匀, 新哈希值的计算方式可以采用公式 *hash*(*m.receiverId* + *localAddress*).

具体算法过程的伪代码描述如下:

算法 1. 分布式消息路由算法
输入: 消息体 msg
输出: 将 msg 转发到所有对应的 session actor
方法:

```
1)  h1 = hash(msg.recv) //计算哈希值
2)  node = binarySearch(h1) //找到环上的下一个节点
3)  forward msg to node
4)  h2 = hash(msg.recv, localAddr) //重新计算哈希值
5)  lr = binarySearch(h2) //找到对应的 LocalRouter
6)  forward msg to LocalRouter
7)  IF userChannelNotExsist(msg.recv) THEN
8)      createUserChannel(msg.recv)
9)  forward msg to UserChannel
10) forward msg to all active sessions
```

4 系统设计与实现

根据前文的分析与设计, 我们实现了一种基于 Actor 模型的分布式 XMPP 服务器. 系统使用 Scala 语言开发, 运行在 JVM 平台上, 使用 Akka[13]作为 Actor 模型框架.

如图 5 所示, 系统由上至下, 按功能划分为以下几个模块:

①负载均衡: 将客户端的连接请求转发到合适的服务器节点, 以保证每个服务器节点承担大致相同的负载. 目前采取的是随机挑选的策略.

② 网络 IO 管理: 负责管理客户端的 TCP 长连接、链路加解密, 处理 IO 异常, 以及为上层提供异步非阻

塞的 IO 方式.

③数据格式解析: 先将接受到的字节流解析为 XML 流, 再解析为 XMPP 数据包, 并发送给对应的 XMPP 处理模块.

④用户消息管理: 负责处理消息类的 XMPP 数据包(Message), 包括消息合法性检验, 消息数据持久化等.

⑤ 用户请求处理: 负责处理请求类的 XMPP 数据包(IQ), 包括用户身份验证、资源绑定等.

⑥ 分布式消息路由: 负责将消息转发到对应的节点上的用户消息管理模块.

⑦集群管理: 负责节点间数据通信, 协调节点一致性, 监控集群状态, 检测节点加入、退出和失效.

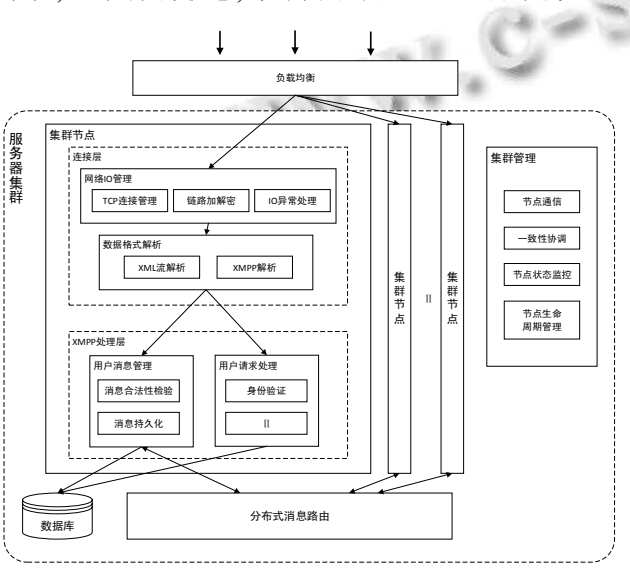

图 5　系统整体架构

整个系统完全基于 Actor 模型设计并实现, 系统中的每个模块都由若干的 Actor 对象组成, 由 Akka 框架负责 Actor 生命周期管理、Actor 间通信、异步任务调度、集群一致性协调等功能.

5　实验验证

本节将对所实现的 XMPP 服务器系统(下文称为 Apus)在不同并发压力下, 就平均消息送达延迟时间和消息成功送达率两方面性能指标进行实验验证. 并选择了目前流行的开源 XMPP 服务器产品 Tigase 系统, 在相同的实验环境下进行对比实验, 并对比分析二者的实验结果.

5.1 实验方案

客户端使用压力测试工具模拟 N 个用户同时登录系统, 待所有用户登录完毕后, 每个用户开始以平均每秒一条消息的速率, 向一个随机选择的用户发送消息, 连续发送 100 条, 再等待 20 秒之后退出. 因此, 在发送消息的过程中, 系统承载的压力是每秒 N 条消息. N 的取值分别为 100、200、500、1000、2000、5000、10000. 最后分别统计并计算两个系统在不同并发压力下的性能指标.

5.2 实验环境

服务端使用两台相同的虚拟机构成集群, 具体配置如下:

处理器	Intel Xeon E5–2620 2.00GHz 四核
内存	4GB
操作系统	Ubuntu 12.04 64bit
JVM	JDK 1.8.0_25

客户端采用压力测试工具 Tsung[14], 版本号为 1.5.2a.

5.3 实验结果及分析

图 6 比较了两个系统在不同并发压力下的平均消息送达延迟时间. 从图中可以看出, 当压力小于每秒 1000 条时, 两个系统的表现都比较好, 延迟时间都在毫秒级别. 当压力超过每秒 1000 条时, Tigase 出现性能瓶颈, 延迟时间急剧增长. 而 Apus 的性能一直保持在较好的水平, 具备良好的弹性扩展能力. 相比于 Tigase, Apus 的消息送达延迟时间在整体上缩短了 4 至 17 倍.

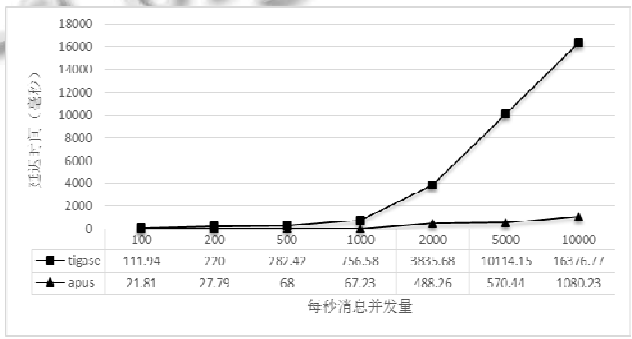

图 6　平均消息送达延迟时间

图 7 展示了两个系统在测试过程中的消息成功送达率. 同样, 当压力超过每秒 1000 条时, Tigase 上大量的消息无法按时送达目标客户端, 系统几乎处于不可用的状态, 而 Apus 的消息成功送达率一直保持在 99.7%以上.

从实验结果可以看出, 依照本文所述方法设计的 Apus 系统, 相比于现有 XMPP 服务器系统 Tigase, 在性能上有大幅度提升, 具备更好的并发处理能力和弹性扩展能力, 可以很好地应对大规模高并发的场景.

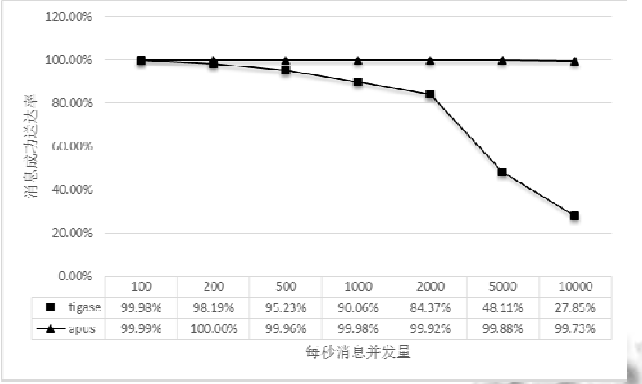

图 7 消息成功送达率

6 结束语

本文首先分析了 XMPP 服务的特点, 总结出设计高性能 XMPP 服务器的关键因素, 并分析现有 XMPP 服务器系统的不足. 然后针对其特点和现有不足, 提出了基于 Actor 模型来设计 XMPP 服务器的方法, 以提高系统的并发能力和扩展能力. 并且对于系统中最重要的路由模块, 设计了一种基于一致性哈希的高效的分布式消息路由算法. 随后, 介绍了系统的设计与实现. 最后通过对比实验表明, 按照本文方法实现的 XMPP 服务器的性能明显优于现有系统, 可以很好地应对大规模高并发的场景.

参考文献

1 刘越.云计算综述与移动云计算的应用研究.信息通信技术,2010,4(2):14–20.

2 李新路.XMPP 协议在 Android 即时通讯系统中的应用.电脑知识与技术,2013,(28):6268–6270.

3 闫龙.基于 XMPP 协议的移动环境下推送系统的设计与实现[学位论文].成都:电子科技大学,2014.

4 Saint-Andre P. Extensible messaging and presence protocol (XMPP): core RFC3920. Internet Engineering Task Force, 2004.

5 张彦,夏清国.Jabber/XMPP 技术的研究与应用.科学技术与工程,2007,7(6):1032–1035,1039.

6 潘凤,王华军,苗放等.基于 XMPP 协议和 Openfire 的即时通信系统的开发.计算机时代,2008,(3):15–16,19.

7 Apache Vysper. http://mina.apache.org/vysper-project/.

8 Tigase XMPP Server Default Clustering Strategy. https://projects.tigase.org/projects/tigase–server/wiki/Default_clustering_strategy/.

9 Ejabberd. https://www.ejabberd.im/.

10 Hewitt C, Bishop P, Steiger R. A universal modular actor formalism for artificial intelligence. Proc. of the 3rd International Joint Conference on Artificial Intelligence. Morgan Kaufmann Publishers Inc. 1973. 235–245.

11 Agha G. Actors: A Model of Concurrent Computation in Distributed Systems. 1985.

12 Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proc. of the 29th Annual ACM Symposium on Theory of Computing. ACM. 1997. 654–663.

13 Akka. http://akka.io/.

14 Tsung. http://tsung.erlang–projects.org/.