RFID 网络中发现服务系统的一种形式化规约

李信鹏[1,3], 赵 文[2,3], 张世琨[2,3], 王立福[2,3]
(1. 北京大学信息科学技术学院, 北京 100871; 2. 北京大学软件工程国家工程研究中心, 北京 100871;
3. 北京大学信息科学技术学院软件研究所高可信软件技术教育部重点实验室, 北京 100871)

摘 要: 本文针对 EPCIS 发现服务在大规模应用和数据安全方面的需求, 基于"集中式索引"模式给出了 EPCIS 发现服务系统的一种体系结构. 首先, 使用 TIOA 语言从高层对整个系统基本功能进行了规约. 然后, 为了使系统在实际应用中负载均衡、具有高性能和可扩展性, 又给出了一种基于 Pub/Sub 的多服务器分布式系统体系结构, 并对该分布式系统的各个组成分别进行了规约. 最后, 本文基于"模拟关系"验证了该分布式系统满足(实现)了高层系统规约.

关键词: 无线射频识别 (RFID); EPCIS 发现服务; 模拟关系
中图分类号: TP311 文献标识码: A 文章编号: 0372-2112 (2008) 12A-009-10

A formal Specification of EPCIS Discovery in the RFID Network

LI Xin-peng[1,3], ZHAO Wen[2,3], ZHANG Shi-kun[2,3], WANG Li-fu[2,3]
(1. *School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*;
2. *National Engineering Research Center for Software Engineering, Peking University, Beijing* 100871, *China*;
3. *Key Laboratory of High Confidence Software Technologies (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871, *China*)

Abstract: Given requirements upon large-scale applications and data security, a kind of architecture of EPCIS Discovery system is proposed based on the "centralized indexing" mode. This paper first provides a high-level specification for the system's basic functionality and requirements using the TIOA language. Further, this paper proposes a Pub/Sub based, distributed multi-server architecture for the system to make it load balancing, highly available, and scalable in practical applications, and then formally specifies every component in the distributed system respectively. Finally, this paper makes use of a simulation relation to show that the distributed system satisfies/implements the high-level specification.

Key words: radio frequency identification (RFID); electronic product code information service (EPCIS); discovery; simulation relation

1 引言

1.1 背景

无线射频识别(Radio Frequency Identification, RFID)技术是一种非接触、多目标、移动目标识别的自动识别技术, 是当前国内外热门的研究课题之一. RFID 技术广泛应用在食品安全、商品防伪、物品召回、物流资产(集装箱、车辆等)管理、仓储管理、公路铁路交通监管、机场行李管理、流水线生产自动化、门禁系统等. 电子物品码(Electronic Product Code, EPC)[1], 是一种在 RFID 标签中记录的电子编码, 用来唯一标识一件物品. RFID 网络是由全体 EPC 参与者组建的一个完整的、复杂的、可扩展的网络. 一个 EPC 参与者是一个业务实体, 它本身向 RFID 网络发布 EPC 数据并且希望获取其他 EPC 数据. 生产商在每件物品里都置入一个 RFID 标签, 每个标签都写入了唯一的 EPC. 通过 RFID 网络, 生产商可以跟踪他的物品, 比如从本地仓储到批发商, 再到零售商的货架等等. 对这些物品感兴趣的其它业务实体(如零售商)也能够访问 RFID 网络, 从而对这些物品进行跟踪或追溯[1].

1.2 RFID 网络的组成

RFID 网络可以分成多个层次, 自低层到高层依次为读写器和中间件(Readers, Middleware)、EPC 信息服务(EPC Infomation Service, EPCIS)、对象名解析服务(Object Naming Service, ONS)、EPCIS 发现服务(EPCIS Discovery)[1]. 下面对这几个主要组成部分作简要介绍.

收稿日期: 2008-07-05; 修回日期: 2008-10-10
基金项目: 国家高技术研究发展计划(863) (No. 2006AA04A119, No. 2006AA04A121); 国家自然科学基金(No. 60803014)

(1) 读写器和中间件

RFID 读写器负责读取 RFID 标签上的编码; 中间件按照不同的业务规则对读到的 RFID 标签进行过滤、收集、分组, 最终产生应用级事件(Application Level Events, ALE).

(2) EPC 信息服务

EPCIS 由 EPCIS Capturing 和 EPCIS Query Server 两部分组成. EPCIS Capturing 是一种应用程序, 它根据预先定义的事件周期规约(Event Cycle specification, EC-spec)[2], 识别和验证 EPC 相关的业务事件的发生, 并且将这些业务事件作为 EPCIS 事件发送到 EPCIS Repository 存储. EPCIS Query Server 为本地以及远程的应用程序提供了查询其内部 EPCIS 事件的统一接口.

(3) 对象名解析服务

ONS 负责将一个 EPC 解析为对应的 EPCIS 服务或者其它服务的地址, 这些服务由 EPC 的拥有者(通常是物品的生产商) 管理和维护.

(4) EPCIS 发现服务

ONS 只是用来获取 EPC 的拥有者(通常为生产商) 所维护的 EPCIS 服务地址, 但是在供应链中, 其他企业的 EPCIS 也可能捕获了与该 EPC 相关的物品流动信息, 而通过 ONS 不能获取这些 EPCIS 服务的地址. 这种服务由 EPCIS Discovery 提供. 在多个参与者组成的供应链中, 通常参与者事先不知道如何访问其他参与者的 EPCIS, 也不可能自己跟踪供应链, 这时就需要使用 EPCIS Discovery. 这正是 RFID 网络所要达到的目标.

1.3 问题的提出和本文的目的

EPCIS Discovery 是 RFID 网络体系结构的一个重要组成部分, 是其中的一个核心服务, 但是现在国内外尚处在对发现服务模式的探索阶段, 还没有建立适应大规模产业应用的发现服务系统. 国内外许多组织正在进行 EPCIS Discovery 的研究, 例如 EPC 网络全球化标准的推动者 EPCglobal, 还有 IBM、中科院自动化所、北京大学等.

鉴于 EPCIS Discovery 在 RFID 应用中的重要作用, 业界也希望能够尽快建立可以实际运行的发现服务系统. 本文通过对 EPCIS Discovery 的需求分析, 给出了一种体系结构, 并对该结构进行形式化规约, 旨在明晰发现服务系统的各项功能, 希望对国内外发现服务系统的开发提供一定参考.

1.4 TIOA 简介

本文选用 Timed Input/Output Automata(TIOA)[3] 对 EPCIS Discovery 的一种体系结构进行了形式化规约. TIOA 是对系统进行规约的一种形式化语言, 适用于对分布式系统建模; TIOA 能够对系统在不同抽象层次上建模, 能够刻画系统要满足的性质; 在 TIOA 中, 通过在不同抽象层次的模型之间建立模拟关系(simulation relation), 能够验证较低层次的模型是否与较高层次的模型具有一致性. 文献[4] 已经使用 TIOA 对 EPCIS 系统进行了形式化规约和验证.

2 相关研究工作

目前, 国内外对 EPCIS Discovery 的研究主要集中在发现服务的模式和实现上, 发现服务的模式共分为以下三种: 集中式仓库型(Centralized Warehouse)、集中式索引型(Centralized Indexing)[5] 和跟踪供应链型(Follow the Chain)[6].

(1) 集中式仓库型

这种模式中, 全局有一个中央的 warehouse. 物品在流经供应链上各个环节时, 所产生的 EPCIS 事件一方面在本地存储, 一方面直接写入 warehouse. 这个过程可通过两种方式完成: 一是 warehouse 主动到各个 EPCIS 上搜集新的 EPCIS 事件, 二是 EPCIS 主动向 warehouse 报告新的 EPCIS 事件. 因此, 用户直接查询 warehouse, 以 EPC 为输入, 即可得到物品在供应链中移动的详细信息.

该模式的优点是实现简单, 用户接口也很简单, 查询速度快. 但是存在局限性, 最重要的问题是, warehouse 如何有效的存储如此海量的数据, 而且还能有效的响应海量的访问. 另外一个是安全问题, EPCIS 对于本地的数据应该是绝对的控制, 尤其是对于复杂的多级安全(例如对于不同级别的用户返回不同程度的信息), warehouse 模式无法做到这一点. 所以, 该模式适用于 EPCIS 数据完全共享、数据规模小、访问量小的供应链环境.

(2) 集中式索引型

这是 EPCglobal 提出的一种改进模式: 全局有一个中央的 Discovery Server(DS). 物品在流经供应链上各个环节时, 产生的 EPCIS 事件, 一方面在本地存储, 一方面以索引(EPC, EPCIS 地址, 时间戳, 其他信息) 的形式向中央的 DS 报告. 用户以 EPC 为输入向 DS 发出查询, 返回一系列的 EPCIS 地址(这些 EPCIS 存有与该 EPC 相关的 EPCIS 事件), 然后用户自行访问各个 EPCIS, 整合成最终结果, 即物品在供应链中移动的详细信息.

该模式的优点一是实现简单, 二是安全. EPCIS 只公开 EPCIS 事件的索引信息, 用户对详细信息的访问完全由 EPCIS 进行访问控制, 可以根据用户不同的安全级别返回相应的信息; 三是中央 DS 负担也较轻: 因为 DS 存储的是索引而不再是完整的 EPCIS 事件, 而且数据库查询负担也相应减轻. 该模式的局限性在于用户接口复杂, 用户需要查询 DS 和各个 EPCIS, 查询 DS 速度较快, 但加上查询各个 EPCIS 的时间, 总体响应速度变慢.

所以, 该模式适用于 EPCIS 数据部分共享的大规模的供应链环境.

(3) 跟踪供应链型

跟踪供应链型是一种分布式的模式, 以 IBM 正在研究的 RFID Traceability: Theseos 为代表. Theseos 作为一种查询引擎安装在各个 EPCIS 上, EPCIS 对本地 EPCIS 事件进行绝对的访问控制. 本地 Theseos 接收查询并结合本地数据和本地安全策略给出一个查询结果, 基于该结果, 最初的查询被重写(可能会加入本地的某些结果数据) 并发往其他 EPCIS 的 Theseos, 如此递归的查询下去. 这个过程称为"Process and Forward". 在各个 EPCIS 上的查询结果也是递归的返回, 最后整合为物品在供应链中移动的详细信息返回给初始查询者.

该模式的优点是采用分布式策略取消了中央 DS, 避免了中央查询压力. 但是局限性在于 Theseos 的实现非常复杂; 查询的响应时间较长, 因为一个查询沿着供应链将变成多个查询, 而且在每个 Theseos 处的处理比较复杂. 所以, 该模式适用于 EPCIS 完全控制本地数据、数据规模小、访问量小的供应链环境.

本文的应用环境及需求特点是: EPCIS Discovery 应该适应大规模应用, 适应跨地域, 跨供应链的应用; 供应链各个参与者(企业) 有权保护其内部有关物品 EPCIS 事件的详细信息, 只允许授权用户访问这些信息, 但应当对较低权限的用户公开一些概要信息, 如 EPCIS 事件的索引. 因此, 集中式索引型发现服务模式更具有应用可行性.

3　EPCIS Discovery 系统高层形式化规约

下面, 从系统边界(输入、输出) 和系统功能方面, 给出 EPCIS Discovery 系统高层的形式化规约. 在此之前, 首先将形式化规约使用的主要基本数据类型定义如下.

3.1　基本数据类型定义

(1) Event

Event 表示 EPCIS 捕获的事件, 即 EPCIS 事件. 它封装了事件发生的时间、地点、参与者等信息[7]. 该数据类型用 TIOA 语言可表达为:

```
Event tuple [
    eventTime: Int, recordTime: Null [Int],
    epcList: Seq [String],
    bizStep: Int, disposition: Int,
    readPoint: Int, bizLocation: Int]
```

其中: eventTime 为事件发生的时间; recordTime 为可选项, 表示事件被记录到系统中的时间; epcList 为事件所涉及的所有物品的 EPC 编码的列表; bizStep 为业务步骤标识(BusinessStepID) 用来跟踪业务步骤, 指明该事件是哪个业务步骤的一部分; disposition 为分拣状态标识(DispositionID) 类似 bizStep, 也用来表示事件之间的业务逻辑依赖关系; readPoint 为事件发生的精确的物理位置, 该信息由物理读写器提供, 如: "reader 1234"; bizLocation 为事件发生的地点, 如"Beijing warehouse".

(2) EventAbstract

EventAbstract 是在 EPCIS 中生成的对 Event 的摘要或者概述, 隐藏了一些需要保护的细节信息. 其数据类型定义如下:

```
EventAbstract tuple [
    eventTime: Int,
    epcPattern: String ]
```

EventAbstract 使用"epcPattern" 替代了 Event 中的"epcList", 因为"EPC pattern"(在文献[2] 中定义) 能够以一个很短的字符串表示"epcList" 中所有的 EPC. 而且, EventAbstract 排除了 Event 中的其它数据项, 是为了保护 EPCIS 的某些私密性信息.

(3) EventPointer

EventPointer 在 EPCIS 中生成, 是 EPCIS 向 DS 报告的关键信息, 也是发现服务的用户请求获取的信息. 其数据类型定义如下:

```
EventPointer tuple [
    eventAbstract: EventAbstract,
    ISaddress: String,
    recordTime: Null[Int]]
```

EventPointer 由 EventAbstract、ISaddress (指明 EventAbstract 的来源 EPCIS 地址) 和可选项 recordTime (指明 EventPointer 被记录到 DS 的时间) 组成.

3.2　系统高层形式化规约

EPCIS Discovery 系统的边界如图 1 所示. EPCIS Discovery 系统作为一个整体来说, 有两对输入和输出. 一对是 capture 和 captureAck, 从功能角度讲, capture 表示从 EPCIS 数据库中检测并捕获所有新生成的 Event, 如果系统没有宕机, captureAck 表示系统对捕获的 Event 逐一给出应答(立即给出, 也允许延迟, 但最终要有应答). 在系统内部, 捕获的 Event 被简化为 EventPointer, 然后注册到系统内部的数据库中. 另一对输入和输出是 query 和 reply, 从功能角度讲, query 表示用户以 EPC 作为参数发出查询, reply 表示系统返回与该 EPC 相关的所有 EventPointer.

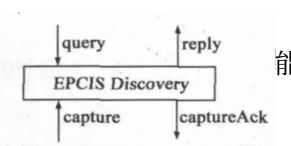


图1　EPCIS Discovery系统的边界

下面对该系统的上述功能需求进行形式化规约:

```
% 系统的高层规约
types
    EPC, % EPC 编码
    Event, % EPCIS 事件
```

```
        EventPointer, % EPCIS 事件的索引
        Exception, % 查询后可能导致的异常
            Reply % 与查询相匹配的所有 eventPointer 或异常
            union[ eventPointers: Seq[EventPointer], exception: Exception]
automaton EPCIS-Discovery
signature
        input capture( e: Event)
        output captureAck( e: Event)
        input query( epc: EPC)
        output reply( epc: EPC, r: Reply)
        % 宕机与恢复, 为系统内部的动作
        internal fail
        internal recover
states
        down: Bool: = false, % 是否宕机
        toAck: Set[ Event]: = {}, % 尚未应答的 Event
        registered: Set[EventPointer]: = {}, % 已经注册的 EventPointer
        queries: Set[ EPC]: = {} % 尚未回答的查询
% generatePointer( e) 将给定的 Event 简化为 EventPointer
let generatePointer( e: Event) = …
% match( eventPointers, epc) 为 true 当且仅当给定的 epc 与给定
% 的 eventPointer 序列匹配
let match( eventPointers: Seq[ EventPointer], epc: EPC) = …
% tag( u) 的参数类型必须是符合类型 union, 用来判断参数属于
% union 中的哪种子类型
let tag( u: uinon) = …
transitions
        input capture( e)
            eff if - down then
                registered: = registered ∪ { ep | ep: = generatePointer
                ( e) };
                toAck: = toAck ∪ { e}
            fi
        output captureAck( e)
            pre ¬ down ∧ e ∈ toAck
            eff toAck: = toAck - { e}
        input query( epc)
            eff if ¬ down then queries: = queries ∪ { epc} fi
% 如果 r 的类型是 EventPointer 序列, 那么它们应当是
%“ registered” 中所有与查询的 epc 相匹配的 EventPointer
% r 的类型也可以是异常
        output reply( epc, r)
            pre ¬ down ∧ epc ∈ queries ∧
                ( tag( r) = eventPointers ⇒ match( r. eventPointers, epc)
                ∧ r. eventPointers ⊆ registered)
            eff queries: = queries - { epc}
        internal fail
            eff down: = true
        internal recover
            pre down
            eff down: = false
% tasks 规约自动机中所有“ query” 动作都对应一个“ reply” 动作
% 所有“ capture” 动作都对应一个“ captureAck” 动作
tasks
        { query( epc: EPC), reply( epc: EPC, r: Reply) }
        { capture( e: Event), captureAck( e: Event) }
```

4 一种基于 Pub/Sub 的分布式 EPCIS Discovery 系统

本文针对上述 EPCIS Discovery 系统提出了一种分布式的体系结构, 主要针对“集中式索引型” 模式中的中央 DS 采用多服务器节点, 因为 DS 是系统的核心部分, 负责处理所有的用户查询请求; 在实际应用中它应当负载均衡, 具有高可用性、容错性和可扩展性.

4.1 系统的体系结构

本文将整个 EPC 集合划分成若干个子集, 划分规则可以有多种: 比如按照物品生产商所在的不同地域进行划分, 或者按照物品所属的不同行业进行划分, 等等. 然后, 本文引入概念“ region” 来代表上述划分的一个子集. 每个 DS 节点负责一个“ region”, 叫做“ RegionDS”, 该节点只关心与 region 范围内的 EPC 相关的那些 EventPointer. 在前面的集中式索引模式中, 供应链中每一个 EPCIS 都向中央 DS 推送 EventPointer, 那么在分布式系统中, 每一个 EPCIS 必须知道对它捕获的不同 EventPointer, 需要向哪个 RegionDS 进行推送. 因此, 我们采用基于集中式控制流-分布式数据流(CCF-DDF) 的 Pub/Sub 模式来解决这个问题. EPCIS Discovery 系统的分布式体系结构如图 2 所示.

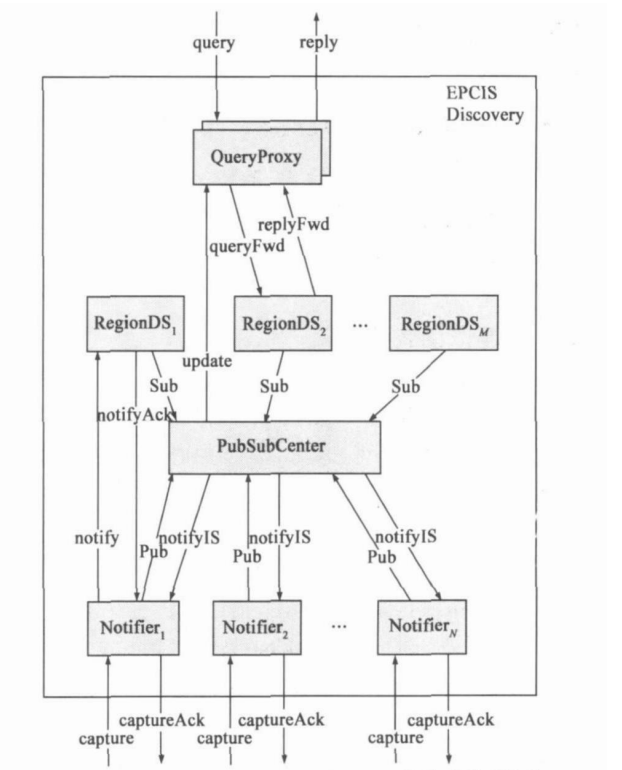


图2 一种基于Pub/Sub的EPCIS Discovery分布式体系结构

该分布式系统由一个 PubSubCenter、M 个 RegionDS、

N 个 Notifier(Notifier 是绑定在各个 EPCIS 上的功能模块)和一个逻辑 QueryProxy 四部分组成, 下面通过描述各部分之间建立的 CCF-DDF 关系和查询流程, 说明各部分功能和该分布式系统的体系结构:

(1) 集中式控制流(CCF)

首先, 所有 RFID 网络的参与者他们所拥有的 EPCIS(Notifier) 分别向 PubSubCenter"发布"他们在实际业务中各自生产或受理的 EPC. 所有 RegionDS 分别向 PubSubCenter"订阅" 他们负责/管理的 EPC. 发布/订阅的 EPC 通常用 EPC pattern 的形式, 因此发布/订阅的条目的数量将大大减少(EPC pattern 不是本文的要点, 所以不作介绍). 然后, PubSubCenter 对发布/订阅的 EPC 进行匹配, 生成一个"三元组"(发布者, 订阅者, 匹配的 EPC pattern) 的列表. 最后, 根据该"三元组" 列表, PubSubCenter 通知每个 EPCIS 一个订阅者列表, 即二元组(订阅者 RegionDSi($1 \leqslant i \leqslant M$), 匹配的 EPC pattern) 列表.

(2) 分布式数据流(DDF)

当物品在供应链各节点发生到达、离开、打包、拆包、加工、或者仅仅是被观察到这些事件时, 每个绑定在 EPCIS 上的 Notifier 分别向它的订阅者(RegionDS) 发送与订阅的 EPC 相关的 EventPointer. 相应的 RegionDS 接收自己订阅的 EventPointer 并存储在内部数据库中, 然后向来源 Notifier 返回应答消息.

(3) 查询流程

首先, PubSubCenter 将所有订阅者(RegionDS) 的 URI 及其订阅的 EPC pattern 发送并周期性的更新到 QueryProxy. 用户以 EPC 为参数向 QueryProxy 发出查询请求, QueryProxy 先在本地查询订阅该 EPC 的 RegionDS 的 URI, 然后, QueryProxy 查询相应的 RegionDS, 并将 RegionDS 返回的结果转交给发起查询的用户.

4.2 系统的形式化规约

4.2.1 Notifier

该分布式系统中的 Notifier 必须向 PubSubCenter"发布"它对应的 EPCIS 在实际业务中生产或受理的 EPC, 并且接收来自 PubSubCenter 的订阅者(RegionDS) 列表, 指明了这些 RegionDS 的 URI 及其订阅的 EPC. 另外, 它还负责将从 EPCIS 捕获的 Event 全部简化为 EventPointer, 并且根据订阅者列表将这些 EventPointer 发送给订阅了该 EventPointer 的 RegionDS, 而不是仅仅发送给"集中式索引型"模式中唯一的中央 DS, 然后等待接收来自相应 RegionDS 的应答. 还有, 它有必要将等待发送的 EventPointer 进行永久存储以保证自身宕机后的状态恢复.

对 Notifier 的上述功能需求进行形式化规约如下:

```
% Notifier 的规约
types
    URI, % 任何服务器的地址
    EPCPattern % EPC pattern 代表一个 EPC 集合
% concern: EPCIS 在实际业务中要受理的所有 EPC
% me: 本地 EPCIS 的 URI
automaton Notifier(concern: Set[EPCPattern], me: URI)
signature
    input capture(e: Event)
    output captureAck(e: Event)
    % 将 EPCPattern 和 EPCIS 的 URI 发布到 PubSubCenter
    output pub(epcPat: EPCPattern, isAddr: URI)
    % 接收由 PubSubCenter 发来的 EPCPattern 和 RegionDS 地址
    input notifyIS(isAddr: URI, rdsAddr: URI, epcPat: EPCPattern)
    % 根据 RegionDS 地址向相应的 RegionDS 推送 EventPointer
    output notify(rdsAddr: URI, ep: EventPointer)
    input notifyAck(ep: EventPointer)
    internal fail
    internal recover
states
    down: Bool := false,
    pending: Set[EventPointer] := {}, % 等待推送的 EventPointer
    toAck: Set[Event] := {},
    subList: Map[EPCPattern, URI] := {}, % 订阅者列表
    pubList: Set[EPCPattern] := concern, % 等待发布的 EPC 集合
let generatePointer(e: Event) = …
% match(ep, epcPat) 为 true 当且仅当给定的 ep 的 epcPattern 与给
% 定的 EPCPattern 匹配
let match(ep: EventPointer, epcPat: EPCPattern) = …
transitions
    input capture(e)
        eff if ¬down then
            pending := pending ∪ {ep | ep := generatePointer(e)};
            toAck := toAck ∪ {e}
        fi
    output captureAck(e)
        pre ¬down ∧ e ∈ toAck
        eff toAck := toAck − {e}
    output pub(epcPat, isAddr)
        pre epcPat ∈ pubList ∧ isAddr = me
        eff pubList := pubList − {epcPat}
    input notifyIS(isAddr, rdsAddr, epcPat)
        eff if ¬down ∧ isAddr = me then
            subList[epcPat] := rdsAddr
        fi
    output notify(rdsAddr, ep)
        pre ¬down ∧ ep ∈ pending ∧
            (∃ epcPat: EPCPattern subList[epcPat] = rdsAddr ∧
            match(ep, epcPat))
    input notifyAck(ep)
        eff if ¬down ∧ ep ∈ pending then
            pending := pending − {ep}
        fi
    internal fail
```

```
        eff down: = true
    internal recover
        pre down
        eff down: = false
tasks
    {capture(e: Event), captureAck(e: Event)}
```

4.2.2 PubSubCenter

PubSubCenter 接收来自所有 Notifier 的发布信息和来自所有 RegionDS 的订阅信息. 然后, 周期性的对发布/订阅的 EPC 进行匹配, 生成一个"三元组"(发布者, 订阅者, 匹配的 EPC pattern) 的列表, 根据"三元组"列表, 将订阅者(RegionDS) 的 URI 和匹配的 EPC pattern 发送给对应的发布者(Notifier). 另外, 它还将所有 RegionDS 的URI 及其订阅的 EPC pattern 发送并周期性的更新到 QueryProxy.

对 PubSubCenter 的上述功能需求进行形式化规约如下:

```
% PubSubCenter 的规约
types
        Time Int, % 整型, 表示时间
        Subscription
            tuple[rdsAddr: URI, % 发起订阅的 RegionDS 地址
                  epcPat: EPCPattern, % 代表 RegionDS 负责/管理的
                  EPC
                  subTime: Time], % 订阅发生的时间
        Publication
            tuple[isAddr: URI, % 发布信息的 EPCIS 地址
                  epcPat: EPCPattern, % 代表 EPCIS 要受理的 EPC
                  pubTime: Time], % 发布发生的时间
        MatchedPubSub %发布信息与订阅信息的匹配结果
            tuple[isAddr: URI, rdsAddr: URI, matchedPat: EPCPattern]
% period: 更新订阅者列表和匹配发布/订阅信息的周期
automaton PubSubCenter(period: Time)
signature
    % 从 EPCIS(或 Notifier) 接收一条发布信息
    input pub(epcPat: EPCPattern, isAddr: URI)
    % 从 RegionDS 接收一条订阅信息
    input sub(epcPat: EPCPattern, rdsAddr: URI)
    % 通知 EPCIS: 某个 RegionDS 订阅了 EPCIS 发布的某些 EPC
    output notifyIS(isAddr: URI, rdsAddr: URI, epcPat: EPCPattern)
    %将订阅者列表发送并周期性的更新到 QueryProxy
    output update(newSub: Set[Subscription])
    % 匹配发布信息与订阅信息
    internal matchPubSub
    internal fail
    internal recover
states
    down: Bool: = false,
    subList: Set[Subscription]: = {}, % 订阅信息的集合
    pubList: Set[Publication]: = {}, % 发布信息的集合
    %发布/订阅匹配结果的集
    matched: Set[MatchedPubSub]: = {},
    now: Time: = 0% 本地时钟
% 返回两个 EPCPattern 的交集
let intersect(epcPat1: EPCPattern, epcPat2: EPCPattern) = …
transitions
    input pub(epcPat, isAddr)
        eff if ¬down then
            pubList: = pubList∪{[epcPat, isAddr, now]}
        fi
    input sub(epcPat, rdsAddr)
        eff if ¬down then
            subList: = subList∪{[epcPat, rdsAddr, now]}
        fi
output notifyIS(isAddr, rdsAddr, epcPat)
    pre[isAddr, rdsAddr, epcPat]∈matched
    eff matched: = matched- {[isAddr, rdsAddr, epcPat]}
%将上一周期内最新的订阅信息向 QueryProxy 更新
output update(newSub)
    pre ¬down∧mod(now, period) = 0∧
        (∀s: Subscription s∈newSub⇒
            now- period< s.subTime⩽now)
internal matchPubSub
    let epcPat: EPCPattern= nil;
    pre ¬down∧mod(now, period) = 0
    %将上一周期内最新的发布信息与所有订阅信息进行匹配
    eff for p: Publication in pubList where
            now-period< p.pubTime⩽now do
        for s: Subscription in subList do
            epcPat: = intersect(s.epcPat, p.epcPat);
            if epcPat≠nil then
              matched: = matched∪{[p.isAddr, s.rdsAddr, epcPat]}
            fi
        od
    od
%将上一周期以前的发布信息与上一周期最新的订阅信息匹配
        for p: Publication in pubList where
            p.pubTime⩽now-period do
        for s: Subscription in subList where
            now-period< s.subTime⩽now do
        epcPat: = intersect(s.epcPat, p.epcPat);
        if epcPat≠nil then
        matched: = matched∪{[p.isAddr, s.rdsAddr, epcPat]}
        fi
    od
od
    internal fail
        eff down: = true
    internal recover
        pre down
        eff down: = false
```

4.2.3 RegionDS

RegionDS 是系统最重要的部分, 负责处理所有的用户查询请求. 每个 RegionDS 只关注与本"region"范围内

的 EPC 相关的那些 EventsPointer, 它向 PubSubCenter“订阅”自己负责/管理的那些 EPC. 当它关注的 EPC 物品在供应链中移动时, 它将接收到发布过这些 EPC 的供应链参与者(Notifier)向它推送的与这些 EPC 相关的 EventPointer, 并将这些 EventPointer 永久存储. 另外, 对于用户查询, 如果服务器没有宕机, 它将回答每一个查询, 返回用户的是与查询的 EPC 匹配的所有 EventPointer 的列表或者返回“server busy”或“timeout”等异常信息.

对 RegionDS 的上述功能需求进行形式化规约如下:

```
% RegionDS 的规约
types
    Exception, % 查询后可能导致的异常
    Reply % 与查询相匹配的所有 eventPointer 或异常
        union[ eventPointers: Seq[ EventPointer], exception: Excep-
        tion]
% incharged: RegionDs 负责/管理的所有 EPC
% me: 本地 RegionDS 的 URI
automaton RegionDS( inCharged: Set[EPCPattern], me: URI)
signature
% 将 EventPointer 插入数据库
    input notify( rdsAddr: URI, ep: EventPointer)
    output notifyAck( ep: EventPointer)
    % 基于给定的 EPC 查询数据库
    input queryFwd( epc: EPC, rdsAddr: URI)
    output replyFwd( epc: EPC, reply: Reply)
    output sub( epcPat: EPCPattern, rdsAddr: URI)
    internal fail
    internal recover
states
    db: Set[EventPointer] := {}, % 存储 EventPointer 的数据库
    toAck: Set[ EventPointer] := {}, % 尚未应答的 EventPointer
    queries: Set[ EPC] := {}, % 尚未回答的查询
    subList: Set[ EPCPattern] := inCharged, % 尚未订阅的 EPC
    down: Bool := false
let match( eventPointers: Seq[ EventPointer], epc: EPC) = …
transitions
    input notify( rdsAddr, ep)
        eff if ¬ down ∧ rdsAddr= me then
            db := db∪{ep};
            toAck := toAck∪{ep}
        fi
    output notifyAck( ep)
        pre ¬ down ∧ ep∈ toAck
        eff toAck := toAck- {ep}
    input queryFwd( epc, rdsAddr)
        eff if ¬ down ∧ rdsAddr= me then
            queries := queries∪{epc}
        fi
    output replyFwd( epc, reply)
        pre ¬ down ∧ epc∈ queries ∧
            tag( reply) = eventPointers= 〉match( reply. eventPoint-
            ers, epc) ∧ reply. eventPointers⊆ db)
        eff queries := queries- {epc}
    output sub( epcPat, rdsAddr)
        pre ¬ down ∧ epcPat∈ subList ∧ rdsAddr= me
        eff subList := subList- {epcPat}
    internal fail
        eff down := true
    internal recover
        pre down
        eff down := false
```

4.2.4 QueryProxy

QueryProxy 作为所有用户查询请求的代理, 它必须保证对用户每个查询 DS 的请求, 必有一个来自 DS 的响应, 所以需要使用一个等待查询的集合 pendingQry. 它还接收 PubSubCenter 周期性的发送过来的订阅者列表(RegionDS 的 URI, 订阅的 EPC pattern)的更新信息, 这样, QueryProxy 本地就维护了一个订阅者列表. 因此, 对每个查询, QueryProxy 可以在本地发起一个 getRDS 动作, 用来获取负责该查询的 EPC 的 RegionDS 地址, 然后将查询请求转发给相应的 RegionDS, 最后将 RegionDS 返回的相关 EventPointer 转发给初始用户.

对 QueryProxy 的上述功能需求进行形式化规约如下:

```
% QueryProxy 的规约
automaton QueryProxy
    Signature
        % 接收 PubSubCenter 周期性发来的订阅者列表更新信息
        input update( newSub: Set[ Subscription])
        % 来自用户的查询和返回查询结果
        input query( epc: EPC)
        output reply( epc: EPC, reply: Reply)
        % 在本地查询负责该 EPC 的 RegionDS 地址, 为内部动作
        internal getRDS( epc: EPC)
        input getRDSAck( epc: EPC, rdsAddr: URI)
        % 将查询转发到相应的 RegionDS, 并接收 RegionDS 的回答
        output queryFwd( epc: EPC, rdsAddr: URI)
        input replyFwd( epc: EPC, reply: Reply)
        internal fail
        internal recover
    states
        subList: Set[ Subscription] := {}, % 周期性更新的订阅者列表
        pendingQry: Set[ EPC] := {}, % 刚收到的未做任何处理的查询
            % 已获得 RegionDS 地址但尚未查询相应的 RegionDS
            toQueryRDS: Set[ tuple[ EPC, URI]] := {},
            replies: Set[ tuple[ EPC, Reply]] := {}, % 需要返回给用户
的回答
            down: Bool := false
    % match( epc, epcPat) 为 true 当且仅当给定的 epc 与 EPCPattern 匹配
```

```
let match(epc: EPC, epcPat: EPCPattern) = …
transitions
    input update(newSub)
        eff if ¬down then subList := subList ∪ newSub fi
        input query(epc)
        eff if ¬down then pendingQry := pendingQry ∪ {epc} fi
    internal getRDS(epc)
        pre ¬down ∧ epc ∈ pendingQry ∧
            (∃ s: Subscription s ∈ subList ∧ match(epc, s.epcPat))
        eff pendingQry := pendingQry − {epc};
            toQueryRDS := toQueryRDS ∪ {[epc, s.rdsAddr]}
    output queryFwd(epc, rdsAddr)
        pre ¬down ∧ {[epc, rdsAddr]} ∈ toQueryRDS
        eff toQueryRDS := toQueryRDS − {[epc, rdsAddr]}
    input replyFwd(epc, reply)
        eff if ¬down then replies := replies ∪ {[epc, reply]} fi
    output reply(epc, reply)
        pre ¬down ∧ {[epc, reply]} ∈ replies
        eff replies := replies − {[epc, reply]}
    internal fail
        eff down := true
    internal recover
        pre down
        eff down := false
tasks
{query(epc: EPC), reply(epc: EPC, reply: Reply)}
```

5 验证

本文已经从高层对整个 EPCIS Discovery 系统进行了规约,并且对一种基于 Pub/Sub 的分布式系统的各个组成分别进行了规约. 那么,该分布式系统是否与前面的高层系统具有相同的外部可见行为,该分布式系统的规约是否满足系统的高层规约,因此需要对该分布式系统进行验证. 验证方法是:证明该分布式系统的4个自动机的合成自动机满足前面的高层规约(即自动机 EPCIS-Discovery).

首先,将上述 4 个自动机合成为一个自动机 DistributedDS:

```
% allEPCs: 所有 EPC 的集合
% allRDSs: 所有 RegionDS 的 URI 的集合
% allNotifiers: 所有 Notifier 的 URI 的集合
automaton DistributedDS(allEPCs: Set[EPCPattern],
allRDSs, allNotifiers: Set[URI])
components
    N[i]: Notifier(concern, i)
        where concern ⊂ allEPCs, i ∈ allNotifiers;
    PSC: PubSubCenter;
    RDS[j]: RegionDS(inCharged, j)
        where inCharged ⊂ allEPCs, j ∈ allRDSs;
    QP: QueryProxy
hiding notify, notifyAck, pub, sub, update, queryFwd,
    replyFwd, notifyIS
```

其中动作 notify, notifyAck, pub, sub, update, queryFwd, replyFwd, notifyIS 都是"hiding"的,意思是对该合成自动机而言,这些动作不再是外部动作(input/output)而是变成了内部动作.

然后,在自动机 DistributedDS 和 EPCIS-Discovery 之间建立一个模拟关系. 模拟关系的定义如下:设有自动机 A 和 B, Q_A 和 Q_B 分别为 A 和 B 的状态集合, x_A, y_A 和 x_B, y_B 分别为 A 和 B 的任意状态,则从 A 到 B 的模拟关系 $R \subseteq Q_A \times Q_B$ 满足如下条件:

(1)如果 x_A 是 A 的一个初始状态,则 B 存在一个初始状态 x_B 使得 $x_A R x_B$;

(2)如果 $x_A R x_B$,并且 A 经过一个动作序列 S_A 从 x_A 转移到 y_A,则 B 必然存在一个对应的动作序列 S_B(可以为空序列),使得 B 从 x_B 转移到 y_B,且满足

$$\text{trace}(S_A) = \text{trace}(S_B) \wedge y_A R y_B$$

其中 $\text{trace}(S_A)$ 表示 S_A 的外部可见行为(外部动作).

下面,我们建立从 DistributedDS 到 EPCIS-Discovery 的模拟关系 R,其中 D 表示 DistributedDS, S 表示 EPCIS-Discovery. 通过 R 将 S 中除了 down 之外的全部状态变量 registered, toAck, queries 与 D 中相关的状态变量之间建立起对应关系:

```
simulation relation R from DistributedDS to EPCIS-Discovery:
% 如果 ep 存在于 S.registered 中,则在 D 中该 ep 必然存在于某个
% Notifier 的 pending 中,或者某个 RegionDS 的 db 中;反之亦然
(∀ ep: EventPointer ep ∈ S.registered ⇔
        (∃ i: URI ∈ allNotifiers ep ∈ D.N[i].pending ∨
        ∃ j: URI ∈ allRDSs ep ∈ D.RDS[j].db)) ∧   % (1)
% 如果 e 存在于 S.toAck 中,则在 D 中该 e 必然存在于某个 Notifier
% 的 toAck 中;反之亦然
(∀ e: Event
    e ∈ S.toAck ⇔ ∃ i: URI ∈ allNotifiers e ∈ D.N[i].toAck) ∧   % (2)
% 如果 epc 存在于 S.queries 中,则在 D 中该 epc 必然存在于
% QP.pendingQry 中,或者 QP.toQueryRDS 中,或者某个
% RegionDS 的 queries 中,或者 QP.replies 中;反之亦然
(∀ epc: EPC epc ∈ S.queries ⇔
    (epc ∈ D.QP.pendingQry ∨
    ∃ j: URI ∈ allRDSs[epc, j] ∈ D.QP.toQueryRDS ∨
    ∃ j: URI ∈ allRDSs epc ∈ D.RDS[j].queries ∨
    ∃ r: Reply[epc, r] ∈ D.QP.replies))   % (3)
```

然后,证明上述关系 R 确实是一个模拟关系:

初始状态下, R 中所有的集合都是空集,所以初始状态下 R 成立. 接下来采用"步步对应"的证明方法,即对 DistributedDS 的每一个动作逐一讨论:

(1)如果 D 中所有组成部分和 S 都没有宕机

①$D.N[i].\text{capture}(e)$:该动作将 e 添加到集合 $D.N[i].\text{toAck}$,将 e 简化成 ep 并将 ep 添加到集合 $D.N[i].\text{pending}$. 在 EPCIS-Discovery 中对应的动作序列是 S.

capture(e), 它将 *e* 添加到集合 S.toAck, 将 *e* 简化成 *ep* 并将 *ep* 添加到 S.registered. 显然, 如果两个自动机的起始状态使得 *R* 成立, 那么经过这两个动作之后只有子句(1)和(2)中的状态变量发生变化, 但子句(1)和(2)仍成立从而 *R* 仍成立.

②*D.N[i].captureAck(e)*: 对应的动作是 S.captureAck(*e*). 这两个动作都是将 *e* 从相应自动机的 toAck 集合中去掉. 只有子句(2)发生变化但仍成立.

③*D.N[i].notify(j, ep)*: 如果 $ep \in D.N[i].pending$, 那么该动作将 *ep* 添加到 *D.RDS[j].db*, 并没有从 *D.N[i].pending* 中去掉, 因此对 DistributedDS 的状态没有影响. 在 EPCIS-Discovery 中对应的动作序列是空序列. 只有子句(1)发生变化但仍成立.

④*D.N[i].notifyAck(ep)*: 该动作只有在 *ep* 被添加到某个 RegionDS 的状态变量 *db*(比如 *D.RDS[j].db*)后才发生, 其效果是将 *ep* 从 *D.N[i].pending* 中去掉. 在 EPCIS-Discovery 中对应的动作序列是空序列. 只有子句(1)的右边状态变量发生变化但仍成立.

⑤*D.QP.query(epc)*: 在 EPCIS-Discovery 中对应的动作是 *S.query(epc)*. 这两个动作都是将 *epc* 分别从相应自动机的 *D.QP.pending* 和 *S.queries* 集合中去掉. 只有子句(3)发生变化但仍成立.

⑥*D.QP.reply*(epc, reply): 该动作将包含 *epc* 的一个二元组从 *D.QP.replies* 中去掉. 在 EPCIS-Discovery 中对应的动作是 *S.reply*(epc, reply), 它将 *epc* 从 *S.queries* 中去掉. 只有子句(3)发生变化但仍成立.

⑦*D.QP.update*(newSub): 该动作只是修改了 *D.QP.subLis*, 但是对 *R* 中的状态变量没有影响.

⑧*D.QP.queryFwd*(epc, *j*): 该动作将[epc, *j*]从 *D.QP.toQueryRDS* 中去掉, 并将 epc 添加到 *D.RDS.queries*. 在 EPCIS-Discovery 中对应的动作序列是空序列. 只有子句(3)右边的状态变量发生变化但仍成立.

⑨*D.QP.replyFwd*(epc, *r*): 该动作将 epc 从 *D.RDS.queries* 中去掉, 并将[epc, *r*]添加到 *D.QP.replies*. 在 EPCIS-Discovery 中对应的动作序列是空序列. 只有子句(3)右边的状态变量发生变化但仍成立.

⑩*D.PSC.pub*(epcPat, *i*): 该动作只是修改了 *D.N[i].pubList* 和 *D.PSC.pubList*, 但是对 *R* 中的状态变量没有影响.

(11)*D.PSC.sub*(epcPat, *j*): 与 *D.PSC.pub*(epcPat, *i*)同理.

(12)*D.PSC.notifyIS*(*i*, *j*, epcPat): 该动作只是修改了 *D.N[i].subList* 和 *D.PSC.matched*, 但是对 *R* 中的状态变量没有影响.

(2)如果 *D* 中所有组成部分和 *S* 都宕机了

这种情况下, *D* 和 *S* 中所有的动作对相应自动机的状态都没有影响, 所以 *R* 仍成立.

(3)如果 *D* 中某些组成部分发生宕机

这时 EPCIS-Discovery.down 的状态将无法确定. 尽管发生宕机的部分可以通过它们宕机之前的状态日志恢复原来的状态, 但是这不是本文的要点, 所以在此不作讨论.

综上所述, 得证 *R* 是从 DistributedDS 到 EPCIS-Discovery 的一个模拟关系. 从模拟关系的性质得出, DistributedDS 系统满足(实现)了高层规约 EPCIS-Discovery.

6　结论

本文针对 EPCIS 发现服务在大规模应用和数据安全方面的需求, 基于"集中式索引"模式给出了 EPCIS 发现服务系统的一种体系结构, 并对该结构进行了形式化规约. 首先, 使用 TIOA 语言从高层对整个系统进行了规约, 使之具备各种基本的功能. 然后, 为了使系统在实际应用中负载均衡、具有高性能和可扩展性, 又给出了一种基于 Pub/Sub 的多服务器分布式系统体系结构, 并对该分布式系统的各个组成分别进行了规约. 最后, 本文基于模拟关系验证了该分布式系统满足(实现)了高层系统规约. 基于该分布式系统的体系结构, 我们实现了 DDS(Distributed Discovery Service)系统, 目前应用于酒类防伪, 跟踪产品在生产商、批发商、零售商等供应链环节的移动细节, 追溯产品来源以达到防伪目的; 还应用于志愿者卡管理, 可实时监控志愿者所在岗位, 以及追溯志愿者从事志愿活动的历史记录, 方便对志愿活动的组织和管理. 下一步, 我们需要将 DDS 系统投入更多实际应用中, 使之处理海量的 EPC 标签、EPCIS 事件及其事件索引等数据和海量的用户查询, 检验系统性能, 进行必要的改进; 并以此为基础进一步研究发现服务的服务质量(QoS)模型和控制方法, 使发现服务的服务质量得以量化和评估.

参考文献:

[1] EPCglobal. The EPCglobal architecture framework[S/OL]. http://www.epcglobalinc.org/standards/architecture/architecture_1_2-framework-20070910.2007-09.

[2] EPCglobal. The application level events(ALE) specification version 1.1[S/OL]. http://www.epcglobalinc.org/standards/ale/ale_1_1-standard-core-20080227.pdf, 2008-02.

[3] Garland S J. TIOA User Guide and Reference Manual[R]. MIT CSAIL, Cambridge, MA, 2005-09.

[4] Assiotis M, Mavrommatis P. The EPCglobal Network: A Formal Specification of EPCIS[R/OL]. http://mavrommatis.googlepages.com/report.pdf, 2006-05.

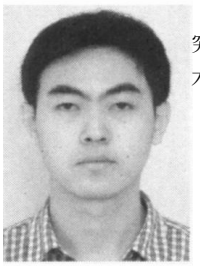
[5] VeriSign. The EPC Network: Enhancing the Supply Chain[R].

VeriSign Inc, Mountain View, CA, 2004.

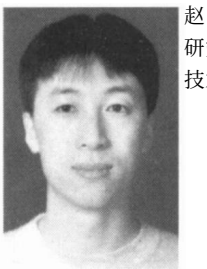
[6] Agrawal R, Cheung A, Kailing K, Schönauer S. Towards traceability across sovereign, distributed RFID databases[A]. Desai B C. Proc. of the 10th Int. Database Engineering & Applications Symposium[C]. Delhi, India, 2006. 174-184.

[7] EPCglobal. EPC information services(EPCIS) version 1.0.1 specification[S/OL]. http://www.epcglobalinc.org/standards/epcis/epcis_1_0_1-standard-20070921.pdf, 2007-09.

作者简介:



李信鹏 男, 1982 年出生, 北京大学博士研究生, 主要研究方向为软件工程和 RFID 相关技术. E-mail: lixp05@sei.pku.edu.cn



赵 文 男, 1967 年出生, 博士, 副研究员, 主要研究领域为软件工程、工作流技术和 RFID 相关技术.