软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(4):790−801 [doi: 10.13328/j.cnki.jos.004753]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种基于元组空间的智能传感器协同感知机制*

王睿智[1,2], 史庭训[1,2], 焦文品[1,2]

[1](北京大学 信息科学技术学院 软件研究所,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 焦文品, E-mail: jwp@sei.pku.edu.cn

摘 要: 随着具有一定计算能力和无线通信能力的智能传感器(称为 mote)的出现,越来越多的物联网应用可以实现对环境及其变化的就地感知、就地决策和就地反应.但因为 mote 的感知能力和感知范围有限,它们需要协同感知才能更全面地感知环境的状态,才能更好地适应环境的变化.而传统的 mote 协同感知的实现方法要求开发人员过多地关注 mote 之间的交互逻辑,并且 mote 应用也无法适应复杂环境的不断变化.为了避免开发人员在交互逻辑上花费过多的精力,同时保障开发出来的 mote 应用系统能够适应不稳定的外界环境,提出了一种基于元组空间的 mote 协同感知支撑机制,使得 mote 之间的协同感知过程(包括交互的建立过程以及对环境变化的适应过程)对开发人员完全透明.最后实现了一个简单的应用场景,展示了该方法如何满足功能需求以及在环境发生变化时如何对环境进行适应.

关键词: 智能传感器;协同感知;自适应;元组空间

中图法分类号: TP311

中文引用格式: 王睿智,史庭训,焦文品.一种基于元组空间的智能传感器协同感知机制.软件学报,2015,26(4):790−801. http://www.jos.org.cn/1000-9825/4753.htm

英文引用格式: Wang RZ, Shi TX, Jiao WP. Collaborative sensing mechanism for intelligent sensors based on tuple space. Ruan Jian Xue Bao/Journal of Software, 2015,26(4):790−801 (in Chinese). http://www.jos.org.cn/1000-9825/4753.htm

Collaborative Sensing Mechanism for Intelligent Sensors Based on Tuple Space

WANG Rui-Zhi[1,2], SHI Ting-Xun[1,2], JIAO Wen-Pin[1,2]

[1](Institute of Software, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

Abstract: With the emergence of intelligent sensors (referred to as motes in this paper) which have certain kinds of computing and wireless communication ability, more and more IoT (Internet of Things) applications can be implemented to sense, judge and react locally according to the environment and its changes. Because of the limitation of motes' awareness and perception scope, they need collaborative sensing to be fully aware of environment states and to better adapt to the changing environment. Traditional mote collaborative sensing mechanisms require developers to focus too much on interaction logics and can hardly adapt to the changing environment. To avoid over spending effort on interaction logics while ensuring that the projects can adapt to the unstable environment, this paper proposes a tuple space based collaborative sensing mechanism for intelligent sensors. It makes communication process completely transparent to developers. This paper also implements a simple demonstration to show how to use this mechanism to build system and how this system adapts to the environment.

Key words: intelligent sensor; collaborative sensing; self-adaption; tuple space

* 基金项目: 国家重点基础研究发展计划(973)(2011CB302604); 国家高技术研究发展计划(863)(2013AA01A605); 国家自然科学基金(913183001, U1201252); 国家创新研究群体科学基金(60821003)

收稿时间: 2014-07-01; 修改时间: 2014-10-14; 定稿时间: 2014-11-14

物联网[1,2]为我们提供了监测和改造客观世界的新手段.物联网应用通过 WSN(无线传感网络,wireless sensor network)中的传感器实时监控并改变物理世界的状态[3],然后,这些传感器只用做基本的输入输出设备,逻辑运算都在后台执行.目前,出现了一种功能更强大的传感器,它们集成了微处理器和通信模块,具有了更强的数据处理能力和无线通信能力.这样的传感器通常叫做智能传感器(intelligent sensor,有些文献中也称其为 smart dust,smart object 或 mote,本文采用 mote 作为这类传感器的代称).Mote 除了拥有比传统传感器更强的计算能力外,还提供了编程接口,使得开发面向物联网前端节点的应用成为了可能.由 mote 构成的系统不同于传统的物联网应用,每个节点不仅仅感知环境,还会对感知到的环境信息进行初级的处理和分析,并依此对环境变化做出反应.此外,mote 节点还可以与其他节点进行通信,实现节点之间的协作,从而可以对环境及其变化的本地化感知、处理和反应.这样的系统相比以往的物联网应用系统具有更多的灵活性,减少了单点失效[4,5]等对系统的影响,可以在极端环境中依然发挥作用,为无线传感器的应用场景提供了更多的可能.

虽然 mote 可以根据需求搭载不同的传感器,但每一个 mote 上面搭载的传感器类型有限,不可能感知全部的环境信息,而且传感器的感知范围有限,获取一个比较大的范围环境信息往往不是单个 mote 可以实现的,因此 mote 之间需要协同感知,以便更准确地感知环境,更好地对环境变化做出反应,适应环境的变化.然而 mote 之间的协同感知也为开发人员带来了新的挑战,一方面是 mote 上的应用程序的计算逻辑往往会和交互逻辑混杂在一起,另一方面,这类应用可能会部署在环境比较恶劣的地方,风雨、电磁等的干扰会影响 mote 的性能,mote 本身也会由于电量受限等原因出现失效,因此协同感知的对象会发生不可预期的变化,如何适应这样的变化也是开发人员不可回避的问题.在以往的实现中,为了确定交互对象并维护交互对象之间的交互过程(关键是维护 mote 之间的数据传递路由),一般使用以下几种方案.

(1) 硬编码地址.在设计系统的时候,把 mote 每个可能的交互对象的地址硬编码在代码中,这样在系统执行过程中,可以直接使用之前定义过的地址进行通信.这种解决方案对于系统节点较少而且交互逻辑简单的系统而言非常直接,开发人员可以对系统有比较清晰的掌控.然而,当系统节点增多或者逻辑复杂之后,每个 mote 的代码中都会有很大一部分的地址定义.而且在填写交互语句的地址时容易出错,正确性只能靠开发人员在编码过程中保证,一旦出现问题,调试极为复杂.此外,这种方案完全无法应对节点失效等变化.因此这种方案只适用于小型且稳定的实验系统,难以在真实系统中发挥效用.

(2) 广播.使用类似无线传感器领域的泛洪路由方法来实现交互.其思路是把所有的消息都以广播的形式发送,传感器如果收到自己负责的语句后执行相关动作,否则以广播的形式继续转发出去.这种方案的实现存在两方面问题:一是能耗比较大,每次消息传播都会附带着很多无用的转播,大大增加整个系统的能耗,这对于能源受限的无线传感器系统是极不现实的;另一方面,为避免广播造成的消息循环发送,每个节点都需要记录每条消息的接受记录,以判断是需要丢弃还是转发,这就需要花额外的精力去维护这样的一个记录.这种方案的好处是当某个节点坏掉以后,和它具有相似功能的 mote 可以很容易地被发现,具有一定的适应性.

(3) 定制路由.由开发人员在设计系统的时候决定如何建立路由.这种方案的性能取决于开发人员选择的路由以及实现的完善程度.如果实现的效果好,可以具有很好的适应性,然而代价是开发人员需要花费很大的精力去实现路由,可能远远大于在计算逻辑上面的开销.

通过分析以上 3 种方案我们发现,传统的方法在解决交互问题时,要么编码的时候需要花费额外的精力在交互逻辑上,要么难以保证系统适应突变的外部环境.为了解决交互过程中的困难,本文借鉴了元组空间的思路.元组空间提供了一种结构化的数据共享方法,并且已有一些相关工作将元组空间实现在无线传感器网络上.然而现有的基于元组空间的方案一般使用广播作为主要手段,而且缺乏对自适应的支持.鉴于这种状况,本文提出并实现了一种基于元组空间的具有自适应性的 mote 协同感知机制.该机制将协同感知分为两类,一类是用来监测环境事件的短效协同感知,另一类是用来监测环境资源的长效协同感知,短效协同感知使用基于广播的事件触发机制,长效协同感知使用定向扩散算法[6,7]自动建立和维护与远程节点之间的数据传递路由.此外,该机制还具有适应环境变化的能力,能够动态发现协同对象,并动态地维护交互路由.本文第 1 节首先介绍元组空间的相关工作及元组空间在 WSN 中的应用.第 2 节介绍本文机制的实现方法.第 3 节通过介绍一个具体的实例说

明如何使用该机制构造系统,并通过对比实验说明该机制在降低代码复杂度和提高适应性方面的作用.最后对本文的工作进行总结.本文的工作是在 Mote Runner[8]的基础上完成的,生成的代码可以通过 Mote Runner 直接部署在物理的 mote 硬件平台上运行.Mote Runner 是 IBM 苏黎世研究院在 2010 年提出的一套面向 mote 的应用开发的集成套件,它提出了一种类似 Java 的开发语言,提供了 mote 的编程环境和运行平台.

1 相关工作

元组空间(tuple space)最早是在 Linda[9]中提出的.元组空间本质上提供了一种数据共享空间,元组(tuple)是元组空间中可共享数据的基本单元,它是一种结构化的数据,可看作面向对象里的对象,由开发人员定义其各个域的含义.比如元组〈“foo”,29〉,第 1 个字段用来标识元组,第 2 个字段表示它所保存的值.元组空间负责管理和维护元组,并向外提供操作元组的接口.Linda 提出了 3 种基本的操作:out(新建一个元组)、rd(读入一个元组)、in(删除一个元组).在 Linda 中,元组空间只有一个,并且可以被所有的进程所访问.进程通过对元组空间进行读写操作从而实现数据的共享.

TeenyLIME[10]是基于元组空间的抽象实现的一个无线传感器中间件.为了适应无线传感器里计算资源的限制,在 TeenyLIME 里没有一个集中的元组空间,而是每个节点维护一个元组空间,并且和 1 跳之内的节点共享数据.其抽象的系统结构如图 1 所示.TeenyLIME 的作者认为,在绝大多数场景中,节点之间的交互都可以在 1 跳之内完成,所以其实现也是基于 1 跳的.TeenyLIME 的核心思想是,某个 mote 如果对某一类事件感兴趣的话,就会在它所有 1 跳范围内的邻居上加载对这个事件的描述(通常是一个数据的区间),当它的邻居感应到的数据位于这个区间内之后,就会向它发起通知,从而触发相应的动作.TeenyLIME 非常适用于开发“感知-反应”型的应用,但是如果两个 mote 之间需要长期且持续的数据传输,而且它们之间距离不止一跳的话,TeenyLIME 的实现就类似于上文提到的广播方案,无法保证较好的性能.

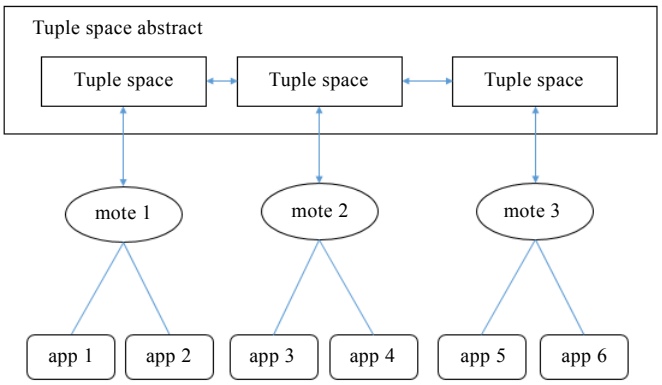


Fig.1 System structure

图 1 系统结构图

Hood[11]的工作使得 mote 可以和 1 跳以内的邻居节点共享数据.在系统设计初期,对于每一个 mote 指定两部分内容:一是它的哪些数据需要共享出去,这样的数据就会持续地被该 mote 广播;另一个是什么样的节点才能作为自己的邻居,只有满足条件的节点发来的数据才会保留在本地 mote,并将其地址加入本地邻居列表.通过这样的设计,每个 mote 都会有多个邻居列表,分别对应自己感兴趣的多种数据.Hood 的基本实现是广播和过滤,而且只在 1 跳之内进行,因此能耗较多,也难以满足多跳的需求.此外,Hood 的系统在编译阶段就确定了共享的数据是什么,不能像 TeenyLIME 一样在运行阶段按需提供数据,这也限制了 Hood 应用的灵活性.

2 基于元组空间的协同感知

2.1 协同感知概念模型

Mote 身处环境之中,其所有行为都可以归结为对环境状态的感知和反应.环境状态包含事件和资源(其中,事件也可以看作一种抽象资源,不过只在某些特定的条件下才会产生).Mote 根据事先设定的规则来判断如何执行自己的行为之前,首先要对环境状态进行感知.然而受制于 mote 感知能力和感知范围的局限性,mote 必须借助其他 mote 的能力来对环境进行协同感知.Mote 在感知事件和资源时的行为有着不同的特点.

Mote 一般不能确定事件的来源和事件产生的时间,因此对事件的感知往往是被动的.感知过程有事件发布者、事件订阅者和事件转发者(非必须).事件订阅者在系统设计之初设定了对事件的反应,直到收到某个事件发布者向环境中发出的事件后才会执行相应的动作.感知过程一般持续时间较短,对数据的可靠性要求较低.

Mote 对资源的感知是主动的.感知过程有资源请求者、资源提供者和资源转发者(非必须).资源请求者向资源提供者主动发起数据请求,资源提供者对数据请求做出反应,资源转发者转发数据请求和数据回应.资源请求者必须事先确定资源提供者的地址,或者至少要确定资源提供者应该满足的条件,然后选择一个最优的资源提供者.感知过程一般持续时间较长,对数据的可靠性有较高的要求.

由以上分析加以总结可得,协同感知过程的概念模型如图 2 所示.环境状态分为事件与资源,每个 mote 根据实际的执行过程在协同感知过程中承担着不同的角色,与事件感知相关的角色有事件发布者、事件订阅者、事件转发者,与资源感知相关的角色有资源请求者、资源提供者、资源转发者.Mote 对环境状态根据角色的不同执行不同的行为.Mote 与环境的交互过程都是通过元组空间来完成的.

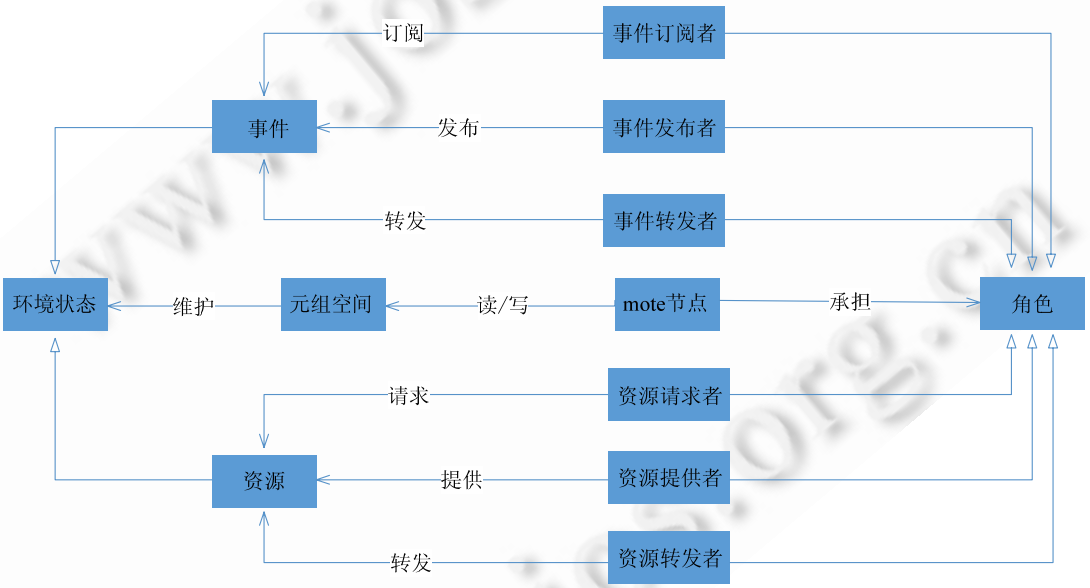


Fig.2 Collaborative sensing concept model

图 2 协同感知概念模型

2.2 协同感知的分类

根据上文中协同感知概念模型的分析,本文将协同感知分为以下两类:

(1) 不确定交互对象的短效协同感知.主要用于“感知-反应”类的应用场景中.比如一个警报器 mote 需要关注火灾事件,当收到火灾事件之后,它会发出警报,并操纵相应的喷水设备.这个火灾事件可能由任意一个携带温度感应器的 mote 在任意时候发出,事件发生的时间和地点都不确定,警报器 mote 只需知道是否发生了某事件即可,而不必知道确切的事件来源.事件处理之后,该交互也就结束了,不需要保存交互的路径.这种类型的协

同感知通常用于处理不确定的突发事件.在有些场景下,若一个 mote 要给多个 mote 发送命令也可以使用这种协同感知,因为这种情况下,广播最直接、迅速,而且相比建立多条交互路径代价也更小.

(2) 确定交互对象的长效协同感知.用于“纯感知”类的应用场景中.在某些应用场景中,交互两端的节点需要在一段时间内持续性地单向或双向输送数据,例如由于硬件限制,无法同时将感应两种数据的感应器安装在同一个 mote 上面,只能由一个 mote *A* 感应亮度,一个 mote *B* 感应温度,但 *B* 需要结合当前的环境温度和亮度做出决策,因此 *B* 就需要持续性地从 *A* 处获取亮度数据.如果每次都使用广播来传播数据,就会造成不必要的能量消耗,尤其是当 *A* 和 *B* 距离不止 1 跳的情况下,位于中间的某些节点就会有不必要的能耗,而这些能耗是可以通过单播方式来避免的.在这样的场景下,最好的方法是先在两个节点之间利用某种路由算法建立路由,然后在建立好的固定路径上通过单播发送数据.

2.3 协同感知的过程及实现

为了实现分布式的 mote 协同感知,我们借鉴了 TeenyLIME 的系统架构,在每一个节点上面维护一个元组空间,通过在元组空间之间传递和使用元组来实现 mote 的协同感知.

2.3.1 短效协同感知

事件订阅者根据应用的需要申明自己感兴趣的事件,在元组空间里插入一个订阅元组,表示对某一类事件感兴趣.事件发布者监测环境变化,当发现环境进入特定的状态时,满足事件的触发条件,就在元组空间中插入一个事件元组,并在系统中广播出去.事件订阅者收到该事件元组后,触发订阅事件元组里定义的反应函数,执行相应的行为.其过程如图 3 所示.

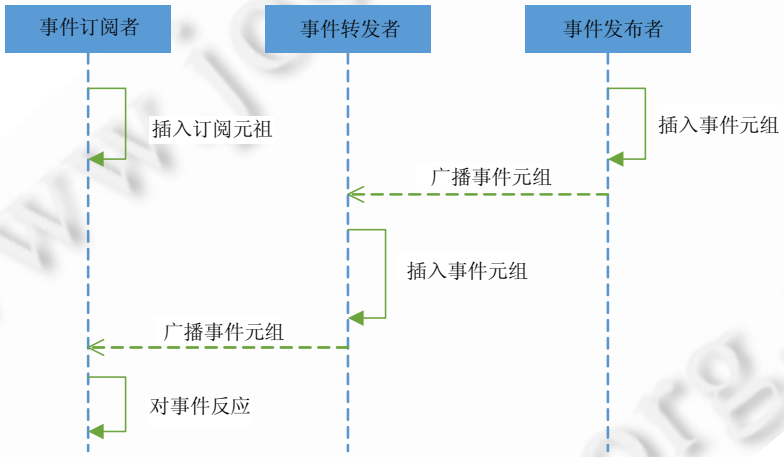


Fig.3 Short-Term collaborative sensing

图 3 短效协同感知过程

2.3.2 长效协同感知过程

(1) 资源提供者在元组空间里创建一个能力元组,表明它们能够对外提供相应的数据.

(2) 当资源请求者需要请求某一类数据时,先在自己的元组空间里查询是否存在该数据,即是否存在相应的资源元组.如果不存在,就广播一个兴趣元组.如果存在,则进入第(7)步获取数据.

(3) 资源转发者在收到资源请求者发出的兴趣元组后,会将该元组继续广播出去.

(4) 如果资源提供者收到了与自己能力元组匹配的兴趣元组,就沿着兴趣元组发来的相反方向单播一个应答元组,该应答元组会记录资源提供者的地址信息.

(5) 当资源请求者收到应答元组后,在自己的元组空间里插入一个资源元组,用来保存应答元组里记录的资源提供者的地址信息.如果同时收到多个应答元组,表明存在多个资源提供者,资源请求者会根据特定的标准(如最短路由、最强信号等)选择其中一个作为协同感知对象,并创建与之对应的资源元组.

(6) 在确定资源提供者之后,资源请求者会沿应答元组的相反方向单播中继元组.中间的资源转发者会将

该元组保存在自己的元组空间中.

(7) 当资源请求者需要数据并且在自己的元组空间中已经存在资源元组时,会根据之前记录的路由信息发送数据请求(消息头为 DATA_REQUIRE 标志),当资源提供者收到数据请求后,会返回数据回应(消息头为 DATA_PROVIDE 标志).资源转发者根据消息头的标志不同将消息转播到正确的方向.

交互对象确定的长效协同感知的详细过程如图 4 所示.

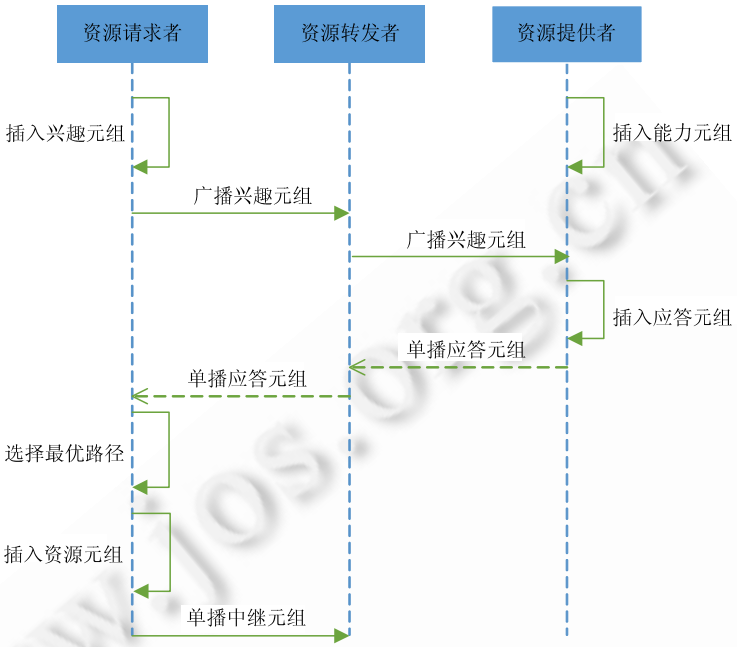


Fig.4 Long-Term collaborative sensing

图 4 长效协同感知过程

2.3.3 协同感知的实现

从上文的描述中可以看出,在 mote 的协同感知过程中,会涉及到多种元组,每个 mote 在本地通过自己的元组空间对它们进行管理元组,其类图如图 5 所示.

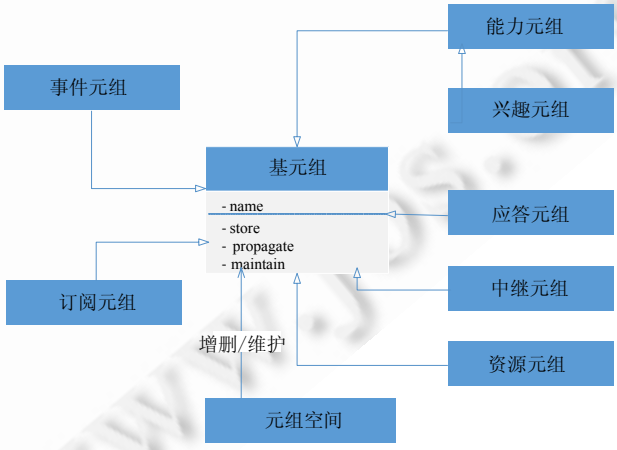


Fig.5 Class diagram

图 5 类图

其中,基元组类是所有元组的基类,其内部有 3 个基本的接口.

(1) store.用来将元组保存在本地元组空间.有些类型的元组无需保存在本地,则不需要实现该接口.

(2) propagate.用于将元组的内容序列化后传播出去.传播的方式可以是广播,也可以是根据某种方式单播.有些元组只需在本地保存一份,则不需要实现该接口.

(3) maintain.用于在本地对元组进行维护.每一种元组都有一些变量需要随时间进行更新,针对不同的元组,maintain 里面有不同的实现策略.元组空间有统一的时钟,每隔一段固定的时间会依次调用所有元组的 maintain 函数,从而实现对元组状态的更新.

元组空间负责对各种元组进行管理和维护.它还提供一些接口,可供开发人员在编写 mote 应用程序时调用.其中最常用的是 inject(byte type,byte[] data),其第 1 个参数指定插入的元组的类型,第 2 个参数是序列化后的元组的数据,元组空间根据第 1 个参数选择特定的构造函数生成不同的元组.紧接着它会先后调用这个新元组的 store 和 propagate 函数,以实现在本地的保存和向外的传播.元组空间还会定期地调用自己的 maintain 函数,该函数通过调用各个元组具体的 maintain 函数来实现对元组的维护.

其他各类元组都是基元组的子类,它们重载了基元组的 store,propagate 和 maintain 方法,因此就具备了不同的行为,并在 mote 协同感知中承担不同角色.下面具体介绍如何使用这些元组来实现上面两种协同感知.

对于不确定交互对象的短效协同感知,会涉及到两类元组.

(1) 事件元组.用来描述一种事件,包含一个名字和一个时效.名字用来唯一标识,时效用来表示可以存在的时间,每次调用 maintain 函数时其时效会减去一点,当时效小于等于 0 时,从元组空间里面删除.当某个事件发布者感应到的数据满足一定的条件时,就会在自己的元组空间里面插入一个事件元组,然后再以广播的方式将它传播出去.每个接收到事件元组的 mote 都在本地元组空间寻找是否有对应的订阅元组,如果有,就触发相应的行为,否则在本地插入一个事件元组,避免循环发送,再将它通过广播的方式传播出去.

(2) 订阅元组.用来表示对某一个事件的订阅,包含一个名字和一个回调函数.名字和事件元组的名字相匹配,回调函数表示当检测到发生该事件以后应该采取的行为.它在事件订阅者的代码里被显示地插入元组空间中,但只保留在本地,不需要传播出去,也不需要时效,因此 propagate 和 maintain 函数都不需要重写.

对于确定交互对象的长效协同感知而言,关键是建立和维护资源请求者和资源提供者之间的路由,在这类协同感知中,涉及 5 种元组.

(1) 能力元组.用来表示提供某种类型的数据的能力,包含一个名字和一个生成函数.名字用来区分不同的元组,比如温度“TEMP”,湿度“HUMI”等等.生成函数用来定义如何产生该数据各个域的内容,比如来自某一个感应器、一个变量表达式或是一个函数的返回值.它也是在资源提供者的应用程序代码中被显式地插入元组空间,不需要传播和时效.

(2) 兴趣元组.用来表示对某一种类型数据的兴趣,包含一个名字和一个时效.名字和某个能力元组相匹配,时效用来控制兴趣元组的存在时间,其原理与事件元组相同.它在 mote 之间通过广播传播.当一个 mote(可能是一个资源提供者,也可能是一个资源转发者)收到一个兴趣元组以后,先在元组空间里寻找是否有匹配的能力元组,如果有,表明它是一个资源提供者,就回应一个应答元组.否则,表明它是一个资源转发者,则在本地插入一个兴趣元组用来避免循环发送.

(3) 应答元组.用来表示对某一种兴趣元组的回应,表明可以提供某一种数据,包含一个名字和一个时效.名字和兴趣元组相匹配,时效的原理与事件元组和兴趣元组相同.当某个 mote 收到一个兴趣元组,并且自己拥有对应的能力元组时,向相反方向单播发送应答元组.应答元组会沿着兴趣元组发来的方向反向单播,直到到达兴趣元组发起的起点为止.

(4) 资源元组.用来记录某种数据的来源地址,包含一个名字、一个源地址和一个状态位.名字与能力元组相匹配.源地址保存数据的来源地址.状态位用于在建立连接的时候判断连接建立到哪个阶段.如果元组空间中存在一个资源元组且其状态为有效,就说明它已经和数据的提供源建立了数据通路.当应答元组到达兴趣元组起点之后,就会在该 mote 插入一个资源元组,并记录其来源地址.

(5) 中继元组.多跳时使用,用于在数据通路的资源转发者转发数据,包含一个名字、一个时效和两个地址.

名字和以上的元组都匹配.地址记录的是消息传播的两端的地址,用于转发消息时寻址.其时效的维护与上述的时效略有不同,将在下文加以介绍.应答元组到达兴趣元组的起点之后,向该路径的相反方向发送中继元组,它会沿着之前应答元组来的相反方向单播回去,并且保存在途经的 mote 中.

2.4 协同感知过程的自适应

正如本文开始部分所分析的,由于外界环境的不稳定性,mote 之间的协同感知需要能够动态地适应环境的变化.短效的协同感知由于使用广播,所以不存在这样的问题.下面简单介绍长效协同感知中如何实现自适应.

(1) 资源请求者请求数据时,调用元组空间的 Read 方法,元组空间会寻找是否有对应的资源元组.

(2) 如果找不到对应的资源元组,返回状态码 *INVALID*,并按照上文所说的流程执行寻址过程.

(3) 如果存在对应的资源元组,判断资源元组的 *waitTime* 的值为多少,如果为 *UNDEFINED*,则向数据来源地址发送一条数据请求,并设 *waitTime* 为 *Tw*;如果不为 *UNDEFINED*,说明仍在等待中,不做任何动作.此时,统一返回状态码 *inProcess*.

(4) 元组空间每隔 *Tt* 时间调用每个元组的 maintain 函数,对于资源元组,如果 *waitTime* 不为 *UNDEFINED*,则 *waitTime* 减 *Tt*.如果 *waitTime* 小于等于 0,说明数据请求超时,该路径失效,删除该资源元组.

(5) 如果元组空间收到资源提供者返回的数据,找到对应的资源元组,保存返回的数据值,并重置 *waitTime* 为 *UNDIFINED*.

(6) 对于资源转发者,中继元组拥有一个域 *Tr* 记录时效,初始为 *Ta*,在每次 maintain 函数被调用时,同样减去 *Tt*,如果 *Tr* 小于等于 0,删除该中继元组.如果收到数据请求或数据回应消息,*Tr* 重置为 *Ta*.

当路径上的某一个节点出现问题后,资源元组会因为等待超时而失效,从而在下一次 Read 动作调用时重建路径,而路径上其他转发节点也会因为太久未收到数据请求或者数据回应消息而使得中继元组的时效小于 0,从而删除无效的路径信息.

以上过程可以实现路径失效的发现和修复,除此以外,本文的机制也便于用户定义各种扩展方法,从而实现路径建立的最优化.在建立路径的时候,兴趣元组的发起端可能收到不止一个应答元组,说明在环境中同时存在多个可用的资源提供者.这时可以设定一定的策略对这些可选的资源提供者进行选择,比如选择距离最短的或者信号强度最强的资源提供者.本文的方法给开发人员提供了一个选择策略的接口,开发人员可以通过重载这个接口来实现最符合应用需求的策略,从而定制个性化的最优策略.

通过以上所说的这些实现,可以在不增加应用本身复杂度的前提下,提高系统的自适应能力.而且这种基于元组空间的实现方案很容易把新的适应策略复用到更多的场景中,开发人员只需选择或者实现一种适应策略,就可以使自己的系统拥有较强的适应能力.

3 实验与分析

3.1 实验场景

为了说明如何使用本文的方法解决实际问题,我们以一个智能实验室的温度调控系统为例展示它在实际场景中的应用.该场景的结构如图 6 所示.控制器可以控制窗户和空调的开关.它周期性地从室内温度感应器读取室内的温度信息.当发现温度超过一定阈值时,从亮度感应器读取亮度信息,如果亮度比较低就认为实验室没有人,不需要作任何调节.否则,从室外温度感应器读取外部温度,当外部温度低于内部温度时,通过开窗通风来降温,否则打开空调调节温度.当温度低于一定阈值时,使用类似的策略进行调节.

在这个场景中,控制器需要和窗户、空调及 3 种感应器分别建立连接,其中和室外温度感应器的通信需要 2 跳,和其他节点的通信需要 1 跳.通过对这个场景的分析可以发现,在该场景中存在如下两类交互.

(1) 短效协同感知.窗户和空调对命令的获取是被动和短效的,所以控制器和它们的交互可以用短效协同的方式来实现.控制器负责判断何时打开窗户或空调,是事件发布者,窗户和空调负责在接到命令之后执行实际的操作,是事件订阅者.在窗户和空调的元组空间里分别插入名为 OPEN_WIN 和 OPEN_AIR 的订阅元组,回调

函数分别是打开窗户和打开空调.控制器通过发布名为OPEN_WIN和OPEN_AIR的事件元组,来对窗户和空调发起打开命令.

(2) 长效协同感知.控制器需要主动地从 3 种感应器中获取环境信息,获取过程持续时间较长,因此它们之间的交互使用长效协同感知.控制器是资源请求者,3 种感应器是资源提供者.此外,由于室外温度感应器不能和控制器直接通信,所以还需要室内温度感应器作为资源转发者.控制器和 3 种感应器需要先建立确定的连接,然后再在路径的基础上进行通信.

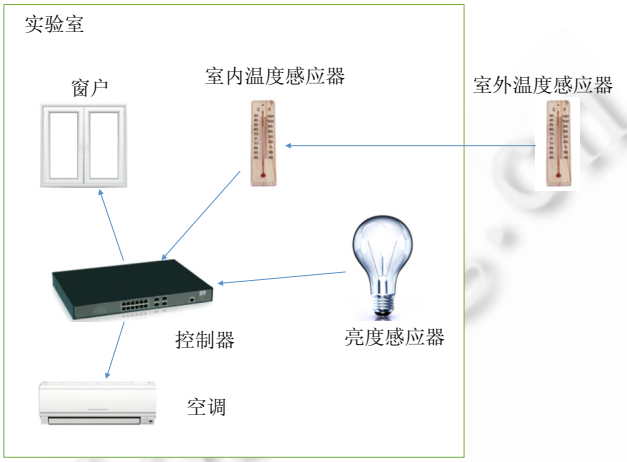


Fig.6 Temperature control system

图 6 温度调控系统结构图

通过以上分析,设置实验组使用本文基于元组空间的实现机制,对照组则使用原始的 Mote Runner 的方案,并且不实现自适应功能.实验结果从它们的代码复杂度和自适应能力两方面进行比较.

3.2 实验结果分析

3.2.1 代码复杂度

图 7 是使用本文机制的实现中应用逻辑的核心代码,图 8 展示了不使用本文方法的室内温度感应器的交互逻辑部分的代码,表 1 是两种方法代码长度的比较.读者可以直观地看出,使用了本文机制的实现要更简洁.产生这些影响的原因是:

(1) 开发者在原始的 Mote Runner 平台上进行开发时,必须获取和维护所有的交互对象的地址,因此就需要一些专门的函数来负责请求的匹配等功能.而使用本文方法的实现,寻找地址的方法在元组空间中已经实现,用户在应用层无需再关注这个过程.

(2) 节点间的交互迫使开发人员必须特别关注消息传播、解析、缓冲等细节,因此把交互部分的逻辑和应用本身的逻辑混杂在一起.而使用本文的方法,通信任务完全由元组空间承担,开发人员只用处理所有和应用相关的数据即可.如本例中,每一种 mote 只要定义可以提供什么数据、需要什么数据以及如何针对这些数据做出判断即可,计算逻辑与交互逻辑完全分离,代码结构变得更加清晰.

(3) 为了保证系统在某些节点失效后能够自恢复,开发人员需要一些额外的状态变量来记录交互对象的状态,通过定期地发送消息来确定交互对象的可用性,进一步加大了交互的复杂度.而本文在实现过程中,在元组空间中实现了一定的自适应策略,并且可以实现一些更复杂的适应策略,复用到更多场景中,而无需开发人员专门维护.

综上原因,使用本文方法的实现可以极大地降低开发人员的编程复杂度,提高编程效率.

```
OuterTemperatureSensor:
    Inject(TupleSpace.Capability,“OUTER_TEMP”)
InnerTemperatureSensor:
    Inject(TupleSpace.Capability,“INNER_TEMP”)
Light:
    Inject(TupleSpace.Capability,“LUX”)
Controller:
    Int outer_temp,inner_temp,light;
    Read(TupleSpace,inner_temp,“INNER_TEMP”)
    if(inner_temp>TEMP_THRESHOLD){
        Read(TupleSpace,light,“LIGHT”)
        if(light>LUX_THRESHOLD){
            Read(TupleSpace,outer_temp,“OUTER_TEMP”){
                if(outer_temp<inner_temp){
                    Inject(TupleSpace.Event,“OPEN_WINDOW”)
                }
                else{
                    Inject(TupleSpace.Event,“OPEN_AIR”)
                }
            }
        }
    }
```

Fig.7 Application logic code in experimental group

图 7 实验组应用逻辑代码

```
InnerTempertureSensor:
…
    If(data[9]==’R’ && data[10]==’E’ && data[11]==’Q’ && data[12]==’1’ && data[13]==’N’ && data[14]==’N’){
        SimpleDevices.read(SimpleDevices.MOTE_LIGHT,1,0,senseData,0,2);
        msg[5]=data[7];
        msg[6]=data[8];
        msg[9]=’R’;
        msg[10]=’E’;
        msg[11]=’P’;
        msg[12]=senseData[0];
        msg[13]=senseData[1];
        radio.transmit(Device.ASAP|Radio.TXMODE_CCA,msg,0,20,0);
    }
else if(data[9]==’R’ && data[10]==’E’ && data[11]==’Q’ && data[12]==’O’ && data[13]==’T’){
        if(data[15]<hops[0]){
            Util.copyData(data,0,msg,0,14);
            Util.set16le(msg,5,Radio.SADDR BROADCAST);
            hops[0]=data[15];
            msg[15]=(byte)(data[15]+1);
            radio.transmit(Device.ASAP|Radio.TXMODE_CCA,msg,0,20,0);
        }
    }
else if(data[9]==’R’ && data[10]==’E’ && data[11]==’Q’ && data[12]==’L’ && data[13]==’G’){
        if(data[15]<hops[1]){
            Util.copyData(data,0,msg,0,14);
            Util.set16le(msg,5,Radio.SADDR_BROADCAST);
            hops[1]=data[15];
            msg[15]=(byte)(data[15]+1);
            radio.transmit(Device.ASAP|Radio.TXMODE_CCA,msg,0,20,0);
        }
    }
…
```

Fig.8 Communication logic code of inner temperature sensor in control group

图 8 对照组中内部温度感应器的交互逻辑代码

Table 1 Comparison of the length of code

表 1 代码长度比较

	对照组	实验组	降低率(%)
温度感应器	99	6	93.9
控制器	161	38	76.4
室内温度感应器	99	6	93.9
室外温度感应器	99	6	93.9

3.2.2 自适应能力

为验证本文方案的自适应能力,通过令室内温度感应器随机时间休眠来模拟节点失效,并增加一个室内温度感应器作为备用节点.正常工作率定义为室内温度感应器正常工作时间占总工作时间的百分比,感知的成功率定义为室内温度的感知成功次数占感知总次数的百分比.图 9 展示了感知成功率与正常工作率之间的变化关系.可以看出,随着正常工作率的降低,没有自适应功能的对照组的感知成功率也随之迅速下降,而使用本文机制的实验组因为可以在一个节点失效之后及时找到备用节点,其感知成功率的下降幅度平缓很多,证明具有一定的自适应能力.

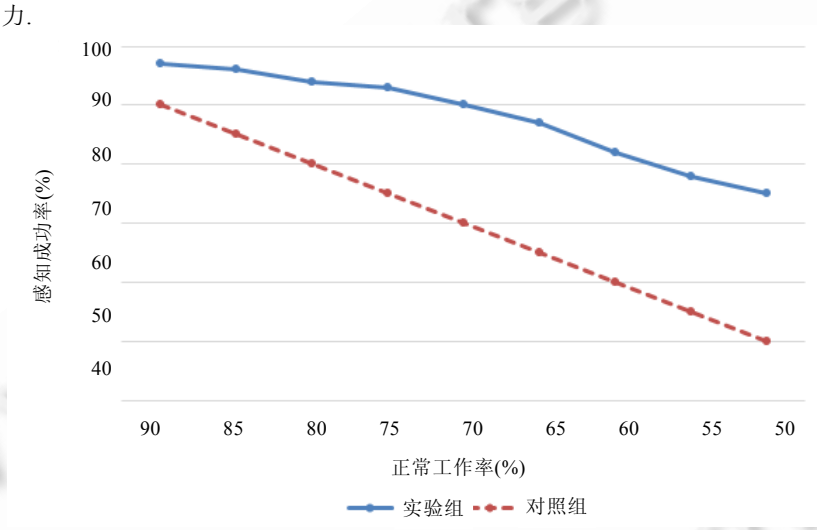


Fig.9 Comparison of adaptability

图 9 适应性比较

除了上面两方面的分析以外,本文方法相比 TeenyLIME 也有所改进.TeenyLIME 主要关注“感知-反应”类的场景,TeenyLIME 的做法是一个节点如果需要监控某一个事件,就会在所有的邻居节点上注册这个事件,如果邻居节点满足触发该事件的条件就通知原节点.然而它的局限在于,事件的注册只限于 1 跳范围内,如果要实现对 1 跳范围之外的事件也能加以反应的话,就需要在中间的转发节点上注册另外一种事件来转发这种消息,在逻辑上不太直观.而对于“纯感知”类的场景,TeenyLIME 的做法是转换成“感知-反应”类的场景来做,如果一个节点发出读取命令的话,会触发 reify 能力元组事件,从而将对应的数据返回给请求方.因为它是基于“感知-反应”类的操作,所以在多跳场景中,仍然存在逻辑不直观的问题.此外,TeenyLIME 并未提及对自适应方面的支持,开发人员仍需自己实现自适应方面的逻辑.

4 总结和展望

通过上述实验场景的分析可以看到,使用本文基于元组空间的协同感知的方法开发系统时,每一个节点都有很清晰的逻辑,开发人员不需要再花费过多精力在通信逻辑上面,可以更投入地进行本地计算逻辑的设计.读取数据过程的完全透明化也使得编写 mote 应用程序可以像编写一般 PC 端程序一样.除此之外,由于本文实现

的元组空间本身拥有自适应机制,所以实现出来的系统也可以很轻松地适应环境的变化,在不稳定的环境下也可以有较好的表现.

在本文工作中,数据的表示还仅仅局限于一些比较简单的数据类型,例如单纯的温度、湿度数据,数据提供方和数据请求方不需要定义比较复杂的数据生成和数据解析逻辑.但在实际应用中,mote 之间可能会传递一些比较复杂的复合数据,一方面可以满足一些更复杂的系统需求,另一方面也可通过合并数据来达到节能的目的.为此,如何更完善地描述数据以及如何将数据描述转换成代码是一个重要的问题.此外,使用本文方法可以把外部的 mote 数据虚拟化为本地的数据,使得不同功能的 mote 可以使用完全相同的代码完成相同的功能.然而,它们在底层实际上还是通过不同的方式实现的,执行的效率也不一样,因此任务代码在不同 mote 之间的动态迁移也是今后研究工作的一个方向.

References:

[1] Neil G, Krikorian R, Cohen D. The Internet of things. Scientific American, 2004,291(4):76−81. [doi: 10.1038/scientificamerican 1004-76]

[2] ITU. The Internet of things. ITU Internet Reports, 2005.

[3] Guo B, Zhang DQ, Yu ZW, Liang YJ, Wang Z, Zhou XS. From the Internet of things to embedded intelligence. World Wide Web, 2012. 1−22. http://link.springer.com/article/10.1007%2Fs11280-012-0188-y

[4] Alcaraz C, Najera P, Lopez J, Roman R. Wireless sensor networks and the Internet of things: Do we need a complete integration. In: Proc. of the 1st Int'l Workshop on the Security of the Internet of Things (SecIoT). Tokyo, 2010.

[5] Zhou L, Chao HC. Multimedia traffic security architecture for the Internet of things. Network, 2011,25(3):35−40. [doi: 10.1109/MNET.2011.5772059]

[6] Intanagonwiwat C, Govindan R, Estrin D. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In: Proc. of the 6th Annual Int'l Conf. on Mobile Computing and Networking (MobiCOM 2000). Boston, 2000. [doi: 10.1145/345910.345920]

[7] Intanagonwiwat C, Govindan R, Estrin D, Heidemann J, Silva F. Directed diffusion for wireless sensor networking. IEEE/ACM Trans. on Networking, 2003,11(1). [doi: 10.1109/TNET.2002.808417]

[8] Caracas A, Kramp T, Baentsch M, Uestreicher M, Eirich T, Romanov I. Mote runner: A multi-language virtual machine for small embedded devices. In: Proc. of the 3rd Int'l Conf. on Sensor Technologies and Applications (SENSORCOMM 2009). IEEE, 2009. [doi: 10.1109/SENSORCOMM.2009.27]

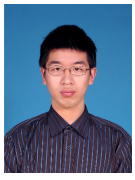
[9] Gelernter D. Generative communication in Linda. ACM Computing Surveys, 1985,7(1). [doi: 10.1145/2363.2433]

[10] Costa P, Mottola L, Murphy AL, Picco GP. Developing sensor network applications using the TeenyLIME: Middleware. Technical Report, DIT-07-059, University of Trento, 2006.

[11] Whitehouse K, Sharp C, Brewer E, Culler D. Hood: A neighborhood abstraction for sensor networks. In: Proc. of the 2nd Int'l Conf. on Mobile Systems, Applications, and Services. 2004. [doi: 10.1145/990064.990079]

王睿智(1991－),男,山西晋城人,硕士,主要研究领域为物联网,自适应软件,软件工程.

焦文品(1969－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,多 agent 系统,自适应软件,软件形式化方法.



史庭训(1988－),男,博士,主要研究领域为物联网,自适应软件,软件工程.