软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2014,25(2):326−340 [doi: 10.13328/j.cnki.jos.004531]

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

一种面向无线传感网应用重编程的逻辑式编程语言*

朱晓瑞, 陶先平, 谢宏伟, 吕 建

(计算机软件新技术国家重点实验室(南京大学),江苏 南京 210023)

通讯作者: 朱晓瑞, E-mail: zxr@smail.nju.edu.cn, http://www.nju.edu.cn

摘 要: 无线传感网的发展,使其需要具有高效地更新其上运行的应用软件的能力.为了解决这个问题,提出了一种面向无线传感网应用重编程的逻辑式编程语言及其处理系统 ReLog.ReLog 语言根据无线传感网应用的普遍特点,基于传统逻辑式语言进行扩展,并提供合适的编程抽象,方便程序员高效地构建、修改程序.同时,语言的处理系统使用中间代码将应用程序与系统软件解耦,从而减少应用更新时所需传输的更新代码的规模,提高更新效率.通过一个数据收集应用案例评估了 ReLog 语言及其执行机制,结果表明:使用 ReLog 语言能够获得简洁、易修改的程序;同时,语言的执行机制能够显著降低传输应用更新代码的能量和时间开销.

关键词: 逻辑式编程语言;操作语义;无线传感网;重编程;解释执行

中图法分类号: TP311 **文献标识码**: A

中文引用格式: 朱晓瑞,陶先平,谢宏伟,吕建.一种面向无线传感网应用重编程的逻辑式编程语言.软件学报,2014,25(2):326−340. http://www.jos.org.cn/1000-9825/4531.htm

英文引用格式: Zhu XR, Tao XP, Xie HW, Lü J. Reprogramming-Oriented logical programming language for wireless sensor network applications. Ruan Jian Xue Bao/Journal of Software, 2014,25(2):326−340 (in Chinese). http://www.jos.org.cn/1000-9825/4531.htm

Reprogramming-Oriented Logical Programming Language for Wireless Sensor Network Applications

ZHU Xiao-Rui, TAO Xian-Ping, XIE Hong-Wei, LÜ Jian

(State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

Corresponding author: ZHU Xiao-Rui, E-mail: zxr@smail.nju.edu.cn, http://www.nju.edu.cn

Abstract: The development of wireless sensor networks requires the ability of efficiently updating application software running on sensors. To address this problem, this paper proposes ReLog, a reprogramming-oriented logical programming language for wireless sensor network applications and its processing system. Based on common characteristics of wireless sensor network applications, the ReLog language extends a traditional logical programming language and provides suitable programming abstractions with which programmers can write and modify programs efficiently. Meanwhile, the language processing system leverages on intermediate code to decouple application programs from system software so as to reduce the size of updating code to improve the updating efficiency. This paper demonstrates ReLog through a case of data collection application. Results show that ReLog can achieve compact and ease-to-modified source code. In addition, the executing mechanism of ReLog can significantly reduce the cost of energy and time when transmitting updating code of applications.

Key words: logical programming language; operational semantics; wireless sensor network; reprogramming; interpreting

无线传感网由资源(包括计算、存储、带宽和能量等)受限的传感器构成,通常被长期部署在目标区域收集数据,以支持各种应用[1].由于部署时间长,无线传感网常因应用维护(如修复 bug)、应用需求变化(如感知新的信息)及应用实现变化(如使用更高效的算法)等原因需要对其传感器上运行的代码进行更新;又因受到部署区域

* 基金项目: 国家自然科学基金(61073031, 61021062); 国家高技术研究发展计划(863)(2012AA011205)
收稿时间: 2013-05-07; 定稿时间: 2013-09-29

和网络规模的限制,更新代码通常需要以无线的方式分发到各传感器上.我们将这种修改源程序、并将编译后的代码更新到传感器上的行为称为重编程(reprogramming)[2].

随着无线传感网应用领域的拓展及其自身规模的增长,重编程对程序设计语言提出了如下需求:

(1) 语言需要提供合适的编程抽象,帮助程序员高效地构建结构合理、易于修改和复用的程序.

(2) 对源程序的修改要易于反映到传感器上,因此语言的执行机制需要为传输的更新代码选择合适的抽象层次.例如,对单个传感器节点来说,传输中间代码所消耗的时间和能量通常仅约为传输二进制镜像的 1%.对于整个网络,传输不同层次代码的时耗和能耗的差距与网络规模成正比[3].

(3) 语言的执行机制还要在网络正常运行开销和重编程开销之间进行权衡.

传统的嵌入式语言[4]不能很好地支持重编程:首先,嵌入式语言未提供合适层次的抽象,方便应用业务逻辑的描述,程序员需要从底层构建应用程序,并依靠经验保证程序结构的合理性.因此,编程工作繁琐,且程序不易修改和复用;其次,由于应用程序与系统软件一起被编译为目标代码,应用程序的更新需要整体替换传感器上的原有代码,更新代价较大.已有许多工作[5−7]针对嵌入式语言抽象层次低、书写程序繁琐的不足提出了不同的解决方法,但是这些工作并没有结合更新传感器上代码的问题一起考虑,因而对重编程的支持仍存在不足.

我们相信,逻辑式语言能够很好地满足重编程的需求.逻辑式编程风范关注“做什么”而不是“如何做”,这种编程风范不仅能够自然地屏蔽实现细节,使程序员更好地关注应用业务逻辑,而且能够降低业务逻辑与编程实现的耦合度,使程序易于修改和复用.同时,逻辑式语言适合解释执行,便于应用代码与系统软件解耦,减少重编程需要传输到传感器上的代码规模,使应用演化灵活.

因此,本文提出了一种面向无线传感网应用重编程的逻辑式语言:ReLog.ReLog 语言的设计沿用了传统逻辑式语言的设计方法,并根据无线传感网应用的特点对语言成分进行扩展.扩展部分的设计充分考虑了无线传感网应用重编程的需求,使 ReLog 语言整体上保持了逻辑式语言低耦合的特性.为了清晰地理解 ReLog 语言,我们详细介绍了语言的各语言成分,并给出了语言中扩展部分的操作语义.为了减少更新代码的规模,ReLog 语言处理系统采用了解释执行的实现方法来将应用代码和系统软件解耦.我们通过实验评估了 ReLog 语言及其处理系统,结果表明:使用 ReLog 语言能够书写简明、松耦合的无线传感网应用程序;同时,其解释执行的方式能够极大地降低应用更新时所需传输的更新代码的规模,从而降低应用更新的时间和能量消耗.

本文第 1 节通过一个示例介绍 ReLog 的总体设计思想.第 2 节给出语言的语法和操作语义.第 3 节简要介绍语言处理系统的设计和实现.第 4 节通过一个应用案例评估 ReLog.第 5 节比较 ReLog 与相关工作.最后一节总结全文,并讨论 ReLog 中的一些问题.

1 ReLog 设计思想

作为许多无线传感网应用的重要组成部分,生成树路由协议能够很好地反映无线传感网应用的特点,其算法描述如下:

(1) 每个节点获取其到一跳邻居的本地链路信息.

(2) 若节点是根节点的一跳邻居,则使用本地链路作为路由,并广播路由信息.

(3) 若节点是根节点的多跳邻居,且收到其一跳邻居广播的路由信息,则利用此邻居节点建立到根节点的路由.路由的代价是节点到其邻居的本地链路代价加上邻居节点到根节点的路由代价.路由建立后,广播路由信息.

(4) 若一个节点发现多条可选路径,则选择代价最小的作为路由.

协议的程序可参见 TinyOS[8]已发布版本中对此协议的实现(程序名为 MultihopLQI).

以此协议的算法为代表分析可知:(1) 无线传感网应用的业务需求可显式地分为数据加工和 I/O 相关操作两个部分.例如,在协议中,计算路由的数据加工过程不关心如何获取数据和如何使用计算结果,“I/O 相关操作”(如收发消息)则负责为“数据加工”提供数据以及处理计算结果.(2) 应用具有事件驱动的特点.例如,协议中节点收到邻居节点路由消息的事件会驱动其计算新路径的代价,并与当前路由做比较.(3) 应用中的数据存在

典型有效期.例如,在协议中,当新路径具有比当前路由更小的代价时,当前路由数据失效;(4) 无线传感网应用节点端业务需求不复杂.

无线传感网应用的特点决定了在为重编程设计语言时应遵循如下原则:

(1) 语言应能在较高层次上提供编程抽象描述数据加工和I/O相关操作,方便应用程序的书写;同时,应通过关注分离降低程序的耦合度.为此,需将数据加工和I/O相关操作显式地分开,并降低两者间的耦合度,使程序易于修改.

(2) 语言的设计应贴合无线传感网应用事件驱动的特点,使语言能够直观地描述无线传感网应用.

(3) 语言需要对应用中数据生存期的典型特征进行抽象,并提供语言设施加以描述.

(4) 语言的设计还应综合考虑其执行机制在无线传感网的特殊环境下对重编程的影响.

基于上述原则,ReLog语言的设计如下:

(1) ReLog语言的语言成分包括子句、shell和属性.子句中的事实和规则用于从高层描述数据加工逻辑;扩展的 shell 中的事件和动作用于建立事件和新事实的联系以及处理推理结果;属性可嵌入子句及shell中,用于声明事实的生存期和以及控制解释执行过程.

(2) 子句部分和shell部分在程序中显式地分开,并且两部分只存在数据耦合.

(3) 为了保持逻辑式语言低耦合的性质,扩展的shell部分中语句相互独立.

(4) 为了高效地将应用更新的代码传输到传感器上,ReLog使用中间代码将应用程序与系统软件解耦.中间代码仅包含与应用业务逻辑相关的代码,因而体积小,便于传输.

图1展示了使用ReLog语言书写的生成树路由协议的样例程序:

```
1.  Clause:
2.      dest(sys_Root).
3.      minCost(sys_NodeID,sys_Root,sys_Infinity)@unique.
4.      path(Source,Dest,Dest,Cost)@volatile(t1):−dest(Dest),link(Source,Dest,Cost).
5.      path(Source,Dest,NextHop,Cost1+Cost2)@volatile(t1):−dest(Dest),link(Source,NextHop,Cost1),
                                        nextHop(NextHop,Dest,Parent,Cost2),Scorce!=Parent.
6.      minCost(Source,Dest,Cost)@unique:−path(Source,Dest,NextHop,Cost),
                                        minCost(Source,Dest,Cost1)@passive,Cost<Cost1.
7.      myNextHop(Source,Dest,NextHop,Cost)@unique:−minCost(Source,Dest,Cost),
                                        path(Source,Dest,NextHop,Cost)@passive.
8.  Shell:
9.      timer(t1,sys_Initial,20000,sys_Inifity){
            getLQI(;Source,Dest,Cost)→insert(link(Source,Dest,Cost)@volatile(t1)).
        }
10.     myNextHop(Source,Dest,NextHop,Cost)→send(1,sys_Broadcast,⟨Source,Dest,NextHop,Cost⟩).
11.     receive(1;⟨NextHop,Dest,Parent,Cost2⟩)→insert(nextHop(NextHop,Dest,Parent,Cost2)@unique).
```

Fig.1 ReLog program of the spanning-tree routing protocol

图1 生成树路由协议的ReLog程序

节点首先获得到达邻居节点的本地链路和代价(第9行).若节点为根节点的一跳邻居,则路由建立(第4行);若节点为根节点的多跳邻居,则通过已经建立路由的邻居节点来建立路由(第 5 行).若存在多条路径,则选择代价最小的那条作为路由(第6行).路由建立后,则广播路由信息(第10行),以便邻居节点收到消息后(第11行)建立路由.

可以看出,ReLog 程序基本上是对协议算法的直接翻译.同时,在协议算法改变时,对源程序的修改也很方便.例如,若协议需要使用不同的链路质量评估策略,则只需将程序第9行的 *getLQI* 替换为需要的策略;又如,若需要在计算路径代价时仅考虑跳数(hop)的影响,则只需将 *path* 中的参数 *Cost* 替换为 *Hops*,并在计算 *path* 的代价时使用此参数(将第5行中的 *Cost*1+*Cost*2 修改为 *Hops*1+*Hops*2).同时,ReLog的解释执行机制使程序修改后需要更新到传感器上的代码量很小.另外,我们基于逻辑式语言进行扩展以及使用解释执行机制的原因,还在于易于在将来实现细粒度的增量式重编程.例如,某次应用更新仅修改了程序中的 1 条规则,则仅将这条规则对应的中间代码更新到传感器上.

2 ReLog 语言成分及其语义

本节首先介绍 ReLog 语言中的各语言成分,随后给出作为扩展成分的属性及 shell 部分的操作语义.

2.1 语言成分

2.1.1 子句定义

ReLog 语言继承了传统逻辑式语言中的许多成分,包括常量、变量、项、谓词、原子、事实和规则:

(1) 常量可以是表示节点编号等的整数,或是表示某些特殊含义的以小写字母开头的标识符,如 *sys_Root*(表示 sink 节点).变量是首字符为大写字母的标识符,如 *Source*.项可以是常量、变量,或是使用项作为参数的函数或算术表达式,如 $Cost_1$+$Cost_2$.若项为常量或变量,则称其为平凡项.

(2) 谓词用于表示对观察对象的性质的判定,使用以小写字母开头的标识符来表示,如 *path*.若 p 为谓词,$t_1,\ldots,t_n$ 为项,则 $p(t_1,\ldots,t_n)$为原子,如 *dest*(*Dest*).特别地,若 $t_1,\ldots,t_n$ 均为平凡项,则称原子 $p(t_1,\ldots,t_n)$为平凡原子;若 $t_1,\ldots,t_n$ 均为常量,则称原子 $p(t_1,\ldots,t_n)$为事实,如 *dest*(*sys_Root*).

(3) 规则由演绎(deduction)符号":–"、符号左边的一个原子(称为规则头)以及符号右边的 1 个或多个平凡原子(称为规则体)组成,如 *path*(*Source*,*Dest*,*Dest*,*Cost*):–*dest*(*Dest*),*link*(*Source*,*Dest*,*Cost*).为了方便使用,ReLog 允许在规则体中使用关系表达式,如 *Source*!=*Parent*,此关系表达式在编译时被翻译为与之等价的内置原子 *sys_neq*(*Source*,*Parent*).

2.1.2 属性定义

除了这些传统的语言成分以外,ReLog 还引入了属性(attribute),用于声明事实的生存期,以及控制解释执行过程.在典型的无线传感网应用中,事实的有效期存在特定的规律.例如,在物体跟踪应用中,描述传感器在每个周期中感知到的物体位置的事实只在当下周期中有效;又如,在路由协议的示例中,每个节点只会保留最新的描述路由信息的 *myNextHop* 事实.因此,程序员可以利用一些属性声明事实的有效期,过期的事实由系统自动删除以节省存储空间.这些属性包括:

(1) @*unique*:在新的事实出现之前,已有的事实持续有效.例如,在路由协议示例中,节点在未找到更好的路由前,已有的用于描述路由信息的 *myNextHop* 事实持续有效.对于具有 *unique* 属性的事实,在更新事实库时 1 次只可向事实库中插入 1 个事实.

(2) @*volatile*(id_{timer}):事实在与之关联的定时器到时触发前有效.例如,*link*(*Source*,*Dest*,*Cost*)@*volatile*(t_1)表示每当定时器 t_1 到时触发时,已有的描述链路质量信息的 *link* 事实全部失效.对于具有 *volatile* 属性的事实,1 次可向事实库中插入多个这样的事实.

与事实有效期相关的属性不能出现在规则体中.

为了节省计算资源,传感器上的解释器需要避免执行一些不需要的规则推理.例如,在示例程序中,当有新的 *path* 数据到来时,在确定由此路径生成的路由比当前路由更优(即,有新的 *minCost* 事实生成)之前,不需要对生成路由的规则(示例第 7 行)进行推理.因此,ReLog 提供了@*passive* 属性:具有 *passive* 属性的原子在其新事实到来时不触发原子所在规则的推理.显然,@*passive* 属性只能出现在规则体中.

2.1.3 Shell 定义

Shell 部分的语句按照功能可划分为插入语句(insertion statement)和执行语句(execution statement).从功能上来看,插入语句用于向事实库插入事实以触发规则推理,执行语句则根据推理出的事实执行相应动作.同时,根据触发语句执行的方式不同,又可分为主动(proactive)语句和反应式(reactive)语句.主动语句由定时器到时触发执行,定时器的定义为 *timer*(*id*,*start*,n_i,n_c),表示定时器 *id* 每隔 n_i ms 触发一次.参数 *start* 表示定时器开始计时的时机:若其值为 *sys_Initial*,则在系统启动时开始计时;若为其他定时器名 *id*′,则在定时器 *id*′到时触发后开始计时.参数 n_c 表示定时器触发的次数:若其值为常数 *n*,则表示只触发 *n* 次;若为 *sys_Infinity*,则定时器会持续工作.反应式语句由相应事件触发.

(1) 插入语句

主动插入语句的形式为

$$timer(id,start,n_i,n_c)\{$$
$$\quad id_{event}(c_1,...,c_n;v_1,...,v_m)\rightarrow insert(predicate(\vec{t}))$$
$$\}$$

其中,$id_{event}(c_1,\ldots,c_n;v_1,\ldots,v_m)$为事件,事件名 id_{event} 和实参 $c_1,\ldots,c_n$ 共同用于注册事件.当定时器到时后,事件发生并将返回结果存放在变量 $v_1,\ldots,v_m$ 中.插入动作 $insert(predicate(\vec{t}))$ 会使用 $v_1,\ldots,v_m$ 中的值将原子 $predicate(\vec{t})$ 初始化为事实,并将其插入到事实库中.例如示例中的第 9 行代码,事件 *getLQI*(;*Srouce*,*Dest*,*Cost*)会将与邻居节点的链路质量信息保存在变量 *Source*,*Dest* 和 *Cost* 中,*insert*(*link*(*Source*,*Dest*,*Cost*))则利用变量的值初始化 *link* 事实并将其插入到事实库中.

反应式插入语句由接收消息事件触发执行,其形式为

$$receive(type_{msg};\langle v_1,...,v_m\rangle)\rightarrow insert(predicate(\vec{t})).$$

当接收到类型为 $type_{msg}$ 的消息时,*receive* 事件发生,消息载荷中的数据被存放在变量 $v_1,\ldots,v_m$ 中.插入动作 $insert(predicate(\vec{t}))$ 会使用 $v_1,\ldots,v_m$ 中的值将原子 $predicate(\vec{t})$ 初始化为事实,并将其插入到事实库中.

(2) 执行语句

主动执行语句的形式为

$$timer(id,start,n_i,n_c)\{$$
$$\quad predicate(v_1,...,v_m)\rightarrow action_1(\vec{v}_1),...,action_n(\vec{v}_n)$$
$$\}$$

其中,$predicate(v_1,\ldots,v_m)$事件用于检查事实库中是否存在谓词 *predicate* 的事实.当定时器到时后事件发生,并选取谓词最新的事实 $predicate(c_1,\ldots,c_n)$中的值 $c_1,\ldots,c_n$ 依次为变量 $v_1,\ldots,v_m$ 赋值.动作 $action_1(\vec{v}_1),...,action_n(\vec{v}_n)$使用 $v_1,\ldots,v_m$ 中同名变量的值作为实参并依次执行.

反应式执行语句由推理出的事实触发,其形式为

$$predicate(v_1,...,v_m)\rightarrow action_1(\vec{v}_1),...,action_n(\vec{v}_n).$$

当事件 $predicate(v_1,\ldots,v_m)$发生时(即,有事实 $predicate(c_1,\ldots,c_m)$被推理出),变量 $v_1,\ldots,v_m$ 会依次使用 $c_1,\ldots,c_m$ 进行赋值.动作 $action_1(\vec{v}_1),...,action_n(\vec{v}_n)$ 会使用 $v_1,\ldots,v_m$ 中同名变量的值作为实参并依次执行.例如示例中的第 10 行代码,当有新的 *myNextHop* 事实产生时,事实中的值会被广播出去.

2.2 操作语义

为了更清晰地理解 ReLog 语言,同时指导语言处理系统的实现,本节首先通过一个示例解释 ReLog 程序的运行过程,然后通过借鉴通信系统演算(calculus of communication system)[9]给出 ReLog 语言的结构化操作语义.由于 ReLog 语言中子句定义部分的语义与传统逻辑式语言基本相同,故这里不再赘述.我们只给出语言扩展部分的操作语义.

2.2.1 ReLog 程序运行过程

首先,通过一个示例描述 ReLog 程序的运行过程,以帮助理解后续将要给出的 ReLog 语言的形式语义.假设应用需要监测环境温度,并要求当温度值高于 37 时,传感器将温度数值连同自己的节点编号广播出去.用 ReLog 语言书写的应用程序的运行过程如图 2 所示.

此程序的运行过程描述如下:

(1) 定时器每 500 ms 到时触发后,感知(sense)事件发生并返回环境温度数值(value);

(2) 与此感知事件关联的 *readings* 原子使用 *Value* 的值初始化为事实;

(3) *readings* 事实被插入到事实库中,

(4) 并触发规则推理;

(5) 推理出的 *warning* 事实同样被添加到事实库中,

(6) 同时,引起 *warning* 事实产生事件发生;

(7) 事实产生事件会触发发送动作,将事实包含的值发送出去.

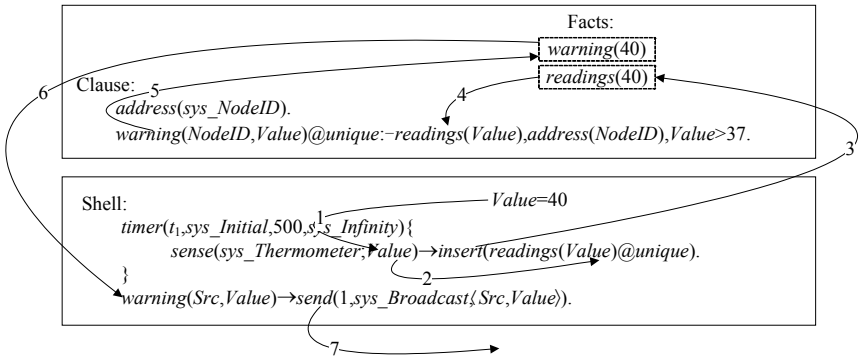


Fig.2 Illustration of the execution of a ReLog program

图 2 ReLog 程序运行示意

在对示例程序中的各语句的语义有了直观了解以后,我们接下来给出语言扩展部分的形式化操作语义.

2.2.2 定义与表示方法

给定序列 S 为 $h::T$,函数 *head* 返回序列 S 的第 1 个元素 h.

给定事实 *fact*,约定 *stamp*(*fact*)表示事实产生的时间.

定义 1. 代入θ为变量及其值的集合:$\theta=\{c_1/v_1,\ldots,c_n/v_n\}$,其中,$v_1,\ldots,v_n$ 为互不相同的变量,$c_1,\ldots,c_n$ 为常量.约定使用符号$\perp$表示无效代入.

给定事件 $id_{event}(c_1,\ldots,c_m;v_1,\ldots,v_n)$,事件发生后将返回一个代入 $\theta=\{c_1'/v_1,\ldots,c_n'/v_n\}$.

给定平凡原子 $p(v_1,\ldots,v_n)$和事实 $q(c_1,\ldots,c_n)$,若谓词 p 与 q 相同,则原子与事实可进行合一(unify).合一的结果为一个代入,其定义为

$$unify(p(v_1,\ldots,v_n),p(c_1,\ldots,c_n))=\begin{cases}\{c_1/v_1,\ldots c_n/v_n\}, & \forall v_i,v_j.(v_i=v_j\rightarrow c_i=c_j)\\ \perp, & \text{o.w.}\end{cases}.$$

给定代入θ和项 $\vec{t}=t_1,\ldots,t_n$, 若对于 $\vec{t}$ 中的任意变量 v,都存在 $c_k/v_k\in\theta$使得 $v=v_k$,则称 $\vec{t}$ 被θ约束,记为 $\theta\downarrow\vec{t}$. 若 $\vec{t}$ 被θ约束,则 $\vec{t}$ 的θ代入 $\vec{t}\theta=c_1',\ldots,c_n'$, 其中, c_i' 是将 $\vec{t}$ 中 t_i 的每个变量 v 替换为常量 $c(c/v\in\theta)$后的计算结果.

给定原子 $p(\vec{t})$ 和代入θ,若 $\vec{t}$ 被θ约束,则 $p(\vec{t}\theta)$ 为事实,并称此事实与原子 $p(\vec{t})$ 关联.

定义 2. 任务可分为推理任务($task_r$)和插入任务($task_i$),推理任务被用于完成规则推理,插入任务被用于向事实库中添加事实.对于规则 $p_h(\vec{t}):-p_1(\vec{v}_1),\ldots,p_n(\vec{v}_n)$, $task_r$ 和 $task_i$ 的定义如下:

$$task_r=\langle p_h(\vec{t}),p_1(\vec{v}_1)::\ldots::p_n(\vec{v}_n)\rangle,$$

$$task_i=\langle p_h(\vec{t}),fact_1^{p_h}::\ldots::fact_n^{p_h}\rangle,$$

其中,$task_r$ 包含规则头 $p_h(\vec{t})$ 以及规则体中平凡原子按源程序中书写顺序组成的序列 $p_1(\vec{v}_1)::\ldots::p_n(\vec{v}_n)$, $task_i$ 包含规则头 $p_h(\vec{t})$ 以及与其关联的事实按产生时间先后顺序组成的序列 $fact_1^{p_h}::\ldots::fact_n^{p_h}$. 对于 $1\leqslant i<j\leqslant n$,

$$stamp(fact_i^{p_h})>stamp(fact_j^{p_h}).$$

给定原子 $p(\vec{t})$ 和代入$\theta_1,\ldots,\theta_n$,且 $p(\vec{t}\theta_1),\ldots,p(\vec{t}\theta_n)$ 为事实,函数 $post_i$ 用于生成插入任务:

$$post_i(\langle p(\vec{t}),\theta_1,\ldots,\theta_n\rangle)=\langle p(\vec{t}),p(\vec{t}\theta_1)::\ldots::p(\vec{t}\theta_n)\rangle.$$

定义 3. 事实库$\mathcal{F}$的定义为 $\mathcal{F}=\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\mid 1\leqslant i\leqslant n\}$, 其中, $p_i(\vec{t}_i)$ 为原子, $S_{fact}^{p_i}=fact_1^{p_i}::\ldots::fact_m^{p_i}$ 为与原子 $p_i(\vec{t}_i)$ 关联的事实组成的序列,且对于 $1\leqslant i<j\leqslant m$, $stamp(fact_i^{p_i})>stamp(fact_j^{p_i})$.

给定事实库$\mathcal{F}$和插入任务 $task_i$,函数 *insert* 用于将 $task_i$ 中的事实插入到事实库中:

$$insert(task_i,\mathcal{F})=\begin{cases}\{\langle p(\vec{t}),fact_1^p::\ldots::fact_n^p\rangle\}\cup\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\}, & \forall(1\leqslant i\leqslant n).p\neq p_i\\ \{\langle p(\vec{t}),fact_1^p::\ldots::fact_n^p::S_{fact}^{p_k}\rangle\}\cup\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\mid i\neq k\}, & \exists(1\leqslant k\leqslant n).p=p_k\end{cases}.$$

给定事实库 $\mathcal{F}=\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\mid 1\leqslant i\leqslant n\}$ 和原子 $p(\vec{t})$, 函数 *exist* 用于判断 $p(\vec{t})$ 是否存在于$\mathcal{F}$中:

$$exist(p(\vec{t}),\mathcal{F})=\begin{cases}\text{true}, & \exists(1\leqslant i\leqslant n).p=p_i\\ \text{false, o.w.}\end{cases}.$$

给定事实库$\mathcal{F}$和原子 $p(\vec{t})$, 若 $exist(p(\vec{t}),\mathcal{F})=\text{true}$, 则函数 *delete* 删除事实库中与 $p(\vec{t})$ 关联的事实:

$$delete(p(\vec{t}),\mathcal{F})=\{\langle p(\vec{t}),null\rangle\}\cup\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\mid p_i\neq p\}.$$

给定原子 $p(\vec{t})$ 和事实库$\mathcal{F}$,函数π返回事实库中与 $p(\vec{t})$ 关联的事实序列 S_{fact}^p, 即, $\pi(p(\vec{t}),\mathcal{F})=S_{fact}^p$.

给定动作 $action_1,\ldots,action_n$,$action_1\triangleright\ldots\triangleright action_n$ 表示依次执行动作 $action_1,\ldots,action_n$.

2.2.3 Shell 部分的操作语义

为了方便讨论,我们首先单独给出定时器的操作语义;然后,在此基础上分别给出 shell 部分中 4 种语句的操作语义.

(1) 定时器的操作语义

如前所述,定时器的定义为 $timer(id,start,n_i,n_c)$.每次定时器到时触发后,均向内部信道β发送一条消息宣布自己触发:

$$\frac{time_{start}\%n_i=0\ \ n_c\neq sys\_Infinity\ \ n_c>0}{timer(id,start,n_i,n_c)\xrightarrow{\beta!id}timer(id,start,n_i,n_c-1)},$$

$$\frac{time_{start}\%n_i=0\ \ n_c=sys\_Infinity}{timer(id,start,n_i,n_c)\xrightarrow{\beta!id}timer(id,start,n_i,n_c)}.$$

其中,$time_{start}$表示定时器开始计时后持续的时间.若参数 *start* 为 *sys_Initial*,则 $time_{start}$=$time_{sys}$,表示定时器在系统启动后持续的时间($time_{sys}$ 为传感器的系统时间);若为其他定时器名,则 $time_{start}$=$time_{sys}$−$time_{\beta?id'}$,表示在信道β接收到定时器 *id*′触发时所发送的消息后持续的时间.

(2) 语句的操作语义

设 S_{task} 表示待执行的任务序列,使用序偶$\langle stm,S_{task}\rangle$表示语句的格局,其表示在当前任务序列下将执行语句 *stm*.各语句的语义如下:

i. 主动插入语句(PIS)

$$\frac{event\rightarrow\theta_1,\ldots,\theta_n\ \ \forall(1\leqslant i\leqslant n).\theta_i\downarrow\vec{t}}{\langle PIS,S_{task}\rangle\xrightarrow{\beta?id_{timer}}S_{task}::post_i(\langle p(\vec{t}),\theta_1,\ldots,\theta_n\rangle)}.$$

当从内部信道β收到定时器到时触发的消息以后,主动插入语句中的事件会发生并返回 1 个或多个代入.随后,*insert* 动作使用这些代入$\theta_1,\ldots,\theta_n$和其包含的原子 $p(\vec{t})$ 向 S_{task} 的末尾添加一个插入任务.

ii. 反应式插入语句(RIS)

$$\frac{type=type_{msg}\ \ event_r\rightarrow\theta\ \ \theta\downarrow\vec{t}}{\langle RIS,S_{task}\rangle\xrightarrow{\alpha?msg}S_{task}::post_i(\langle p(\vec{t}),\theta\rangle)}.$$

当从外部信道α中收到消息(*msg*)时,若消息类型与 *receive* 事件中给定的类型相同,则 *receive* 事件返回一个代入θ.随后,*insert* 动作使用θ和其包含的原子 $p(\vec{t})$ 向任务序列 S_{task} 的末尾添加一个插入任务.

iii. 主动执行语句(PES)

$$\frac{event_{exist}\rightarrow\theta\neq\perp\ \ \forall(1\leqslant i\leqslant m).\theta\downarrow\vec{v}_i}{\langle PES,S_{task}\rangle\xrightarrow{\beta?id_{timer}}action_1(\vec{v}_1\theta)\triangleright\ldots\triangleright action_m(\vec{v}_m\theta)},$$

其中, $\theta=unify(p(\vec{v}),head(\pi(p(\vec{v}),\mathcal{F})))$. 当从内部信道$\beta$收到定时器到时触发的消息以后,主动执行语句中检测与原子 $p(\vec{v})$ 关联的事实是否存在的事件发生,并返回原子与其在事实库中关联的最新事实合一后得到的代入θ.随后,语句中的动作会使用θ对参数赋值并依次执行.

iv. 反应式执行语句(RES)

$$\frac{p_{fact}=p\ event_{gen}\rightarrow\theta\ \forall(1\leqslant i\leqslant m).\theta\downarrow\vec{v}_i}{\langle RES,S_{task}\rangle\xrightarrow{\gamma?fact}action_1(\vec{v}_1\theta)\triangleright\ldots\triangleright action_m(\vec{v}_m\theta)},$$

其中, $\theta=unify(p(\vec{v}),p_{fact}(\vec{c}))$. 当从内部信道$\gamma$接收到推理出的事实时,若事实的谓词与语句中原子的谓词相同,则新事实产生事件发生并返回一个代入θ.随后,语句中的动作会使用θ对参数赋值并依次执行.

2.2.4 属性的操作语义

(1) *@unique* 属性

给定事实库$\mathcal{F}$和插入任务 $task_i=\langle p(\vec{t})@unique,fact\rangle$, *@unique* 属性影响向事实库中插入事实的 *insert* 函数的定义.若 $exist(p(\vec{t}),\mathcal{F})=\text{true}$,则

$$insert(task_i,\mathcal{F})=\{\langle p(\vec{t}),fact\rangle\}\cup\{\langle p_i(\vec{t}_i),S_{fact}^{p_i}\rangle\mid p_i\neq p\}.$$

即,新事实的插入会删除已存在的事实.

(2) $@volatile(id_{timer})$属性

使用序偶$\langle p(\vec{t})@volatile(id_{timer}),\mathcal{F}\rangle$ 表示属性的格局,则$@volatile(id_{timer})$的操作语义为

$$\frac{exist(p(\vec{t}),\mathcal{F})\rightarrow\text{true}}{\langle p(\vec{t})@volatile(id_{timer}),\mathcal{F}\rangle\xrightarrow{\beta?id_{timer}}delete(p(\vec{t}),\mathcal{F}))}.$$

即,定时器到时触发后删除事实库中与原子 $p(\vec{t})$ 关联的所有事实.注意,*volatile* 属性在实现时并未完全按照此语义.若定时器到时触发后解释器正在执行推理任务,则删除操作延迟到推理结束之后进行,以避免 *volatile* 属性的副作用.

(3) *@passive* 属性

给定事实 $p(\vec{c})$ 和规则 $p_h(\vec{t}):-p_1(\vec{v}_1),\ldots,p_m(\vec{v}_m)$, 属性*@passive*影响推理任务的生成.设函数$post_r$用于生成推理任务,其定义为

$$post_r(p(\vec{c}),R)=\begin{cases}null, & (\forall(1\leqslant i\leqslant m).p\neq p_i)\vee(\exists(1\leqslant i\leqslant m).p=p_i\wedge attribute(p_i)=passive)\\ \langle p_h(\vec{t}),p_1(\vec{v}_1)::\ldots::p_m(\vec{v}_m)\rangle, & \exists(1\leqslant i\leqslant m).p=p_i\wedge attribute(p_i)=null\end{cases}.$$

3 语言处理系统

使用嵌入式语言[4]书写的程序会与系统软件一起被编译为二进制镜像.由于二进制镜像通常规模较大,通过无线传输将其分发到传感器上代价高昂;同时,因为二进制镜像中应用程序与系统软件耦合紧密,对镜像中的应用代码进行增量更新存在诸多限制[10].

为了降低更新传感器上代码的代价,ReLog 使用中间代码将应用程序与系统软件解耦.中间代码仅包含与应用业务逻辑相关的代码,因而体积小,便于传输.因此,ReLog 语言处理系统选择了编译加解释的体系结构:包括运行在服务器端的编译器和运行在传感器端的解释器(如图 3 所示).编译器负责将源程序转换为与具体传感器平台无关的中间代码,解释器和系统软件则常驻在传感器上解释执行接收到的中间代码.

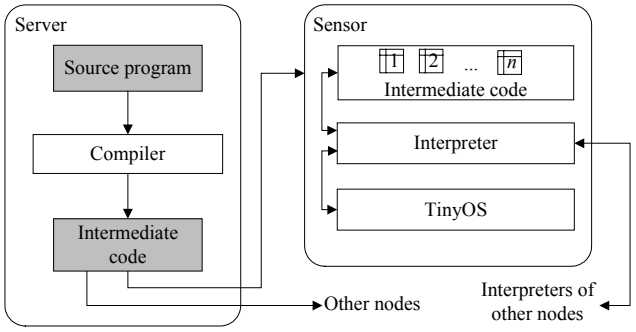


Fig.3 Architecture of the ReLog processing system

图 3 ReLog 处理系统体系结构

3.1 编译器

编译器的主要任务为生成仅包含应用业务逻辑的中间代码,从而减少应用更新所需要传输的代码量,提高重编程效率.同时,考虑到解释执行机制会给传感器执行应用代码带来一定的额外负担,因此,编译器的另一项重要任务是对中间代码进行优化,以提高解释器的计算速度并降低其能耗.

传感器上解释器的计算需求主要源自推理过程,包括:

(1) 当新事实到来时,查询需要进行推理的规则.

(2) 在推理过程中,查找能与当前原子进行合一操作的事实.

(3) 合一操作.其中,前两项查询操作带来了可观的计算开销,因此需要对其进行优化.

我们在中间代码中使用了如图 4 所示的索引图技术来解决这个问题:(1) 每条规则都知道自己的规则体中包含的所有原子;(2) 与原子关联的所有事实按照存在时间递增顺序组织成链表,原子指向链表中第 1 个事实.因此,当有新事实到来时,根据事实关联的原子可知,应选择哪些规则进行推理.同时,在规则推理过程中,规则体中每个原子都可直接找到可用于合一操作的事实.

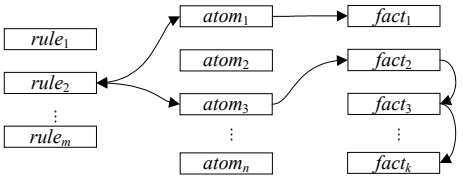


Fig.4 Illustration of the index graph

图 4 索引图示意

3.2 解释器

解释器与支撑其运行的系统软件一起常驻在传感器上解释执行中间代码.解释器使用 nesC[4]语言在 TinyOS[8]操作系统的基础上实现,其核心部分包括调度器、推理器、事实管理器和指令执行器(如图 5 所示).

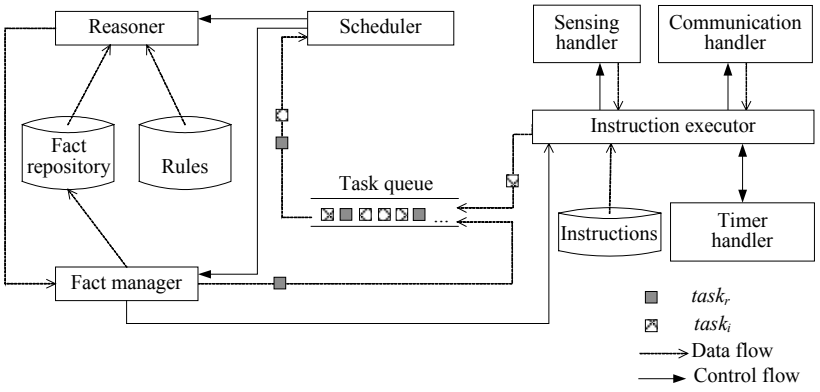


Fig.5 Architecture of the ReLog interpreter

图 5 ReLog 解释器体系结构

调度器负责根据任务类型调用不同的功能模块执行任务:若为推理任务,则调用推理器进行推理;若为插入任务,则调用事实管理器更新事实库.推理器负责对规则进行推理,若推理出新事实,则将事实交予事实管理器.事实管理器将推理出的事实插入到事实库,并根据事实向任务队列中添加相应的推理任务,同时,还将事实推送给指令执行器.指令执行器负责在收集到数据(包括接收到和感知到的数据)以后,向任务队列中添加相应的插入任务,或者根据事实管理器推送来的事实执行相应的动作.

3.2.1 实现选择

(1) 运行机制

解释器实现的一个重要问题是,如何在保证推理正确性的同时及时响应事件.例如在单条规则的推理过程中,推理涉及的事实不应被修改.此时,若事件发生(如接收消息)并请求更新事实库,则此事件应被响应,同时,更新请求应被推迟到规则推理完成后进行(如图6所示).为了解决此问题,解释器的实现使用了TinyOS[8]提供的任务机制:任务之间以非抢占的方式按照发布时间的先后顺序执行,任务执行可以被事件中断.因此,ReLog解释器中的规则推理和事实库更新均被封装成任务.当事件发生时,当前的任务执行会被中断以响应事件,但是事件的处理(更新事实库)则被封装成任务放入任务队列中等待执行.

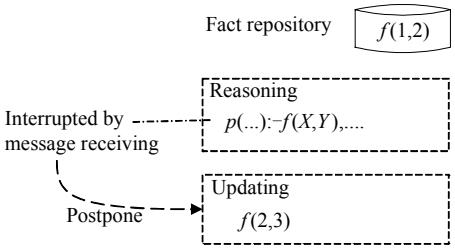


Fig.6 Running mechanism of the interpreter

图 6 解释器运行机制

(2) 内存分配

在无线传感网应用运行过程中,事实库中的事实变化频繁,因此,动态内存分配理论上可更好地支持推理过程和事实库管理.但是,ReLog解释器的实现选择了静态内存分配,其原因如下:

(1) 由于用户界面不友好,收集传感器上程序运行失败的信息很困难.静态内存分配易实现在内存不足时主动通知程序员,帮助程序员修改程序.

(2) TinyOS[8]对动态内存分配的支持不足.与实现健壮的动态内存分配相比,静态内存分配实现代价小,并且可以根据解释器中数据操作的特点进行优化.

4 案例研究

本节使用温度数据收集应用作为案例评估ReLog在如下4个方面的表现:(1) 通过使用ReLog语言书写应用程序展示语言的简明性;(2) 通过比较编译后代码的规模来评估更新的灵活性;(3) 通过比较每个传感器的丢包率和能耗来评估解释器的性能;(4) 通过比较应用程序运行时传感器上的内存开销来评估解释器的适用性.在评估中,我们使用nesC语言书写的程序作为对比目标.

温度收集应用要求传感器每500 ms采集一次温度数据,并将数据连同传感器节点编号一起发送给基站.父节点需要转发其子节点的数据包.为了更清楚地展示解释器的性能,我们在温度收集应用中使用静态路由来消除路由协议(例如CTP协议)带来的影响.应用的网络拓扑如图7所示.

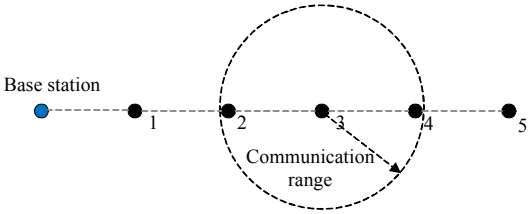


Fig.7 Topology used in the data collection application

图 7 数据收集应用中使用的拓扑

4.1 语言的简明性

图 8 展示了使用 ReLog 语言书写的应用程序,其含义为:每隔 500 ms 定时器到时后,利用温度感知设备采集温度数据生成 *reading* 事实并插入到事实库中(第 6 行),*reading* 事实触发规则推理生成 *log* 事实(第 3 行);若接收到子节点的消息,则生成 *forwarding* 事实并插入到事实库中(第 7 行),*forwarding* 事实也会触发规则推理生成 *log* 事实(第 4 行);推理出 *log* 的事实会触发发送动作的执行,将 *log* 中记录的数据发送给父节点(第 8 行).

```
1.  Clause:
2.      nodeID(sys_NodeID).
3.      log(ID,Temperature)@unique:−nodeID(ID),reading(Temperature).
4.      log(ID,Temperature)@unique:−forwarding(ID,Temperature).
5.  Shell:
6.      timer(t1,sys_Initial,500,sys_Infinity){
            sense(sys_Thermometer;Temperature)→insert(reading(Temperature)@unique).
        }
7.      receive(1;⟨ID,Temperature⟩)→insert(forwarding(ID,Temperature)@unique).
8.      log(ID,Temperature)→send(1,sys_NodeID−1,⟨ID, Temperature⟩).
```

Fig.8 ReLog program of the data collection application

图 8 数据收集应用的 ReLog 程序

使用 ReLog 语言书写的数据收集程序基本上可以看做是对应用需求的简单翻译,并且程序员不用考虑与应用需求无关的诸如硬件设备管理、消息缓冲等实现细节.

4.2 更新的灵活性

图 9 展示了使用 nesC 语言书写的程序在编译后(目标平台为 MicaZ 传感器)的二进制镜像和 ReLog 程序编译后的中间代码的规模对比:二进制镜像为 40 261 字节,而中间代码为 245 字节,仅为二进制镜像的 0.6%.其原因在于:

(1) nesC 程序会与操作系统一起编译,编译后的二进制镜像中大部分代码与应用业务逻辑无直接关系,而 ReLog 中间代码仅包含应用业务逻辑的相关代码.

(2) 与二进制镜像相比,ReLog 中间代码的抽象层次较高.

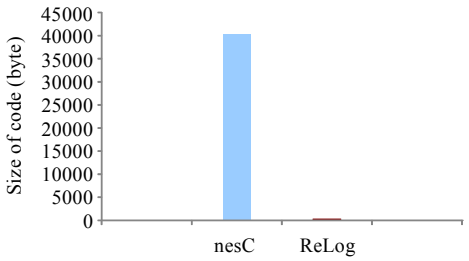


Fig.9 Size of code

图 9 代码规模

为了更直观地展示二进制镜像和中间代码对重编程的影响,我们使用 Avrora[11]模拟器在 MicaZ 传感器上比较传输两种代码的能量和时间开销:

- 在传输二进制镜像时,我们使用载荷为 28 字节(默认最大载荷)的数据包;
- 在传输中间代码时,则使用了 ReLog 用于传输中间代码的真实数据包(载荷为 24 字节).

图 10 和图 11 分别展示了传输两种代码的能耗和时耗:传感器在传输二进制镜像时,共消耗了 8 826mJ 的能量以及 4 251ms 的时间;而在传输中间代码时,仅消耗了 508mJ 的能量以及 225ms 的时间.由此可见,ReLog 可以很好地支持重编程的需求,使应用演化灵活.

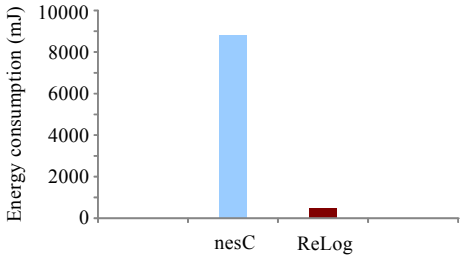


Fig.10 Energy consumption of code transmission

图 10 代码传输的能耗

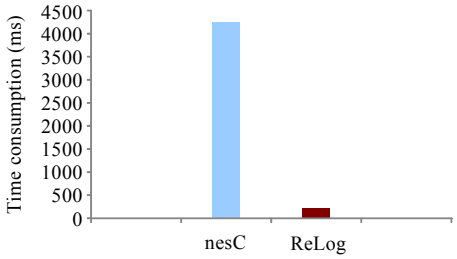


Fig.11 Time consumption of code transmission

图 11 代码传输的时耗

4.3 解释器的性能

我们将 nesC 版本和 ReLog 版本的程序分别在 Avrora 模拟器上运行 60 s.图 12 给出了两个版本的程序运行结果中每个传感器总共发送数据包的数量.与预期一样,越靠近基站的传感器,丢失的数据包越多.但可以看出,两个版本中每个传感器的丢包情况非常接近,因此在这个应用中,ReLog 解释执行的方式可以满足应用需求.

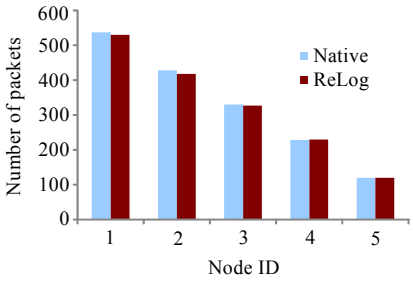


Fig.12 Number of packets sent by each node

图 12 节点发送数据包的数量

图 13 给出了两个版本中每个传感器的能耗情况.ReLog 版本传感器的总能耗在 900mJ~1 286mJ 之间;而 nesC 版本在 882mJ~1 205mJ 之间.ReLog 版本的传感器比 nesC 版本最多多消耗 6.7%的总能量(节点 1).而在每一轮(定时器两次触发之间)中,ReLog 版本的传感器仅比 nesC 版本多消耗 0.15mJ~0.68mJ 的能量,这与一次更新节省的能耗(8 318mJ)相比是可以接受的.

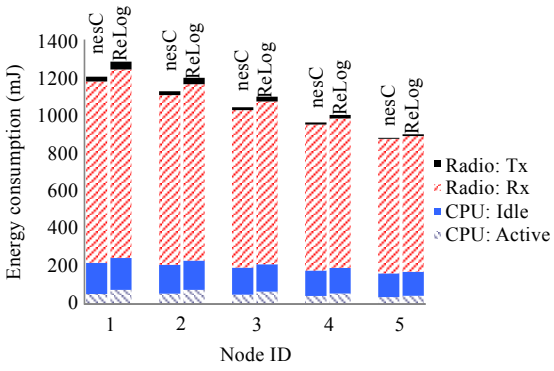


Fig.13 Energy consumption of each node

图 13 节点能耗

另外,在 ReLog 版本中,传感器的通信能耗(尤其是发送消息的能耗)较高的原因在于数据包较大:因为无法预测传输应用数据所需的载荷大小,目前版本的解释器预先将数据包载荷设置为 24 字节.在这个应用中,nesC 版本的程序所使用的数据包载荷仅为 4 字节,因而能耗较低.在未来的工作中,我们将为应用提供多种载荷的数据包,应用可选择最合适的数据包发送数据,以此来进一步降低能耗.

4.4 解释器的适用性

传感器的内存资源非常有限.在流行的传感器平台中,MicaZ 可提供 128KB 的 ROM 和 4KB 的 RAM,而 TelosB 可提供 48KB 的 ROM 和 10KB 的 RAM.图 14 给出了两个版本的程序在 MicaZ 平台上运行的内存开销.ReLog 程序(即解释器)需要 26 366 字节的 ROM 和 3 725 字节的 RAM,而 nesC 程序仅需要 14 322 字节的 ROM 和 931 字节的 RAM.解释器的 RAM 消耗很高,其原因在于解释器使用了静态内存分配,也就是说,对于所有应用程序,其所需要的 RAM 始终约为这么多.另外,由于传感器上通常只运行 1 个程序,因此 RAM 全占用的情况是可接受的.

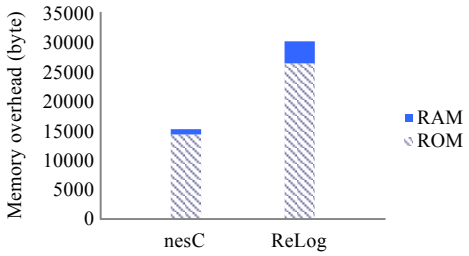


Fig.14 Memory overhead

图 14 内存开销

5 相关工作

随着无线传感网的发展,许多工作致力于为应用编程提供支持,其中一些工作借鉴了申述式方法.

TinyDB[12]和 Cougar[13]将无线传感网看作是关系数据库中的表,并提供类似 SQL 的语言让用户查询数据.查询结果由各传感器节点流向基站,并在汇集过程中根据需要进行合并,以减少传感器的资源消耗.TinyDB 和 Cougar 提供的这种编程抽象能使用户写出非常简洁的代码,但同时,也因为抽象程度过高从而只能支持部分无线传感网应用[14].与这些工作不同,ReLog 语言的设计总结了典型无线传感网应用的普遍特点,使其能够支持大多数无线传感网应用.

Snlog[15]也是逻辑式语言,其在传统逻辑式语言的基础上扩展了位置标识符和内置谓词机制.位置标识符用于声明数据的宿主,以此向用户屏蔽传感器间的通信细节;内置谓词机制可让用户使用嵌入式语言实现某些自定义谓词,通常用于将逻辑式程序与定时、感知等 I/O 相关操作链接起来.Snlog 使用逻辑式语言成分处理无线传感网应用中的数据加工部分,而将逻辑式语言不擅长处理的 I/O 相关操作留给嵌入式语言解决,因而削弱了逻辑式语言所带来的松耦合的特性,增加了源程序修改的难度.同时,Snlog 中的嵌入式语言成分促使语言处理系统采用了将源程序与系统软件一起编译为二进制镜像的实现方式,因而不利于应用更新.与 Snlog 不同,ReLog 语言主要支持无线传感网应用的重编程.为了保持逻辑式语言松耦合的特性,作为扩展的 shell 部分提供高层抽象用于描述 I/O 相关操作;同时,其与子句部分在程序中显式地分开,且两者只存在数据耦合.此外,shell 部分中的语句也相互独立,这样的设计不仅使得 ReLog 程序耦合度低,且语言的处理系统便于使用解释执行机制来降低更新代码的规模,从而高效地支持重编程.

在解释执行方式的相关工作中,VM*[16]在传感器上部署剪裁后的 Java 虚拟机,以解释执行源程序编译后得到的字节码.VM*致力于使用虚拟机来解决无线传感网规模性和传感器节点异构性的问题.虚拟机通常并不提供全部服务,若所要解释执行的代码包含虚拟机未提供的服务,则首先需要对虚拟机进行更新.与 VM*不

同,ReLog 致力于提高重编程的效率.部署在传感器上的解释器提供执行应用代码可能需要的所有服务,因此不需要对解释器进行更新.同时,对于相同的源程序,ReLog 编译器所生成的中间代码规模通常更小.

6 总 结

无线传感网的发展使得重编程问题日益重要.本文设计和实现了 ReLog——一种面向无线传感网应用重编程的逻辑式语言.ReLog 语言基于无线传感网应用的普遍特点,提供合适的编程抽象,方便程序员书写、修改和复用程序.同时,语言的执行机制使用中间代码将应用程序与系统软件解耦,极大地减少了应用更新所需传输的代码量,提高了重编程的效率.

最后,我们就 ReLog 语言及其执行机制的一些问题进行讨论:

(1) 语言成分

程序员在使用 ReLog 语言书写程序时,通过插入语句向事实库插入事实,通过申明原子的属性,让系统自动删除相关事实.ReLog 并未提供语言成分支持程序员显式地删除事实库中的事实,其原因如下:

- 首先,我们使用 ReLog 书写了几类典型的无线传感网应用,发现现有的语言成分已能很好地支持应用编程.
- 其次,事实的不当删除将会造成推理过程出错,提供显式删除事实的语言成分,不利于保证程序的正确性.

(2) 语言的可扩展性

DSN 提供的内置谓词机制,允许程序员使用嵌入式语言实现自定义谓词.ReLog 目前并未提供类似扩展机制,原因如下:

- 首先,对这种扩展机制的不当使用会导致源程序结构复杂,难以修改.
- 其次,扩展部分使用嵌入式语言实现,这提高了将源程序编译为中间代码的难度.

以上两点均不符合 ReLog 作为支持应用重编程语言的目的.

(3) 语言的执行机制

ReLog 语言解释执行的机制在传感器执行应用代码时不可避免地带来一些额外开销,但是我们可以看出:

- 首先,无线传感网应用节点端的业务并不复杂.例如第 1 节所示的生成树路由算法对无线传感网应用来说已较为复杂,但在图 1 所示的 ReLog 程序中,仅使用 7 条语句(4 条规则和 3 条 shell 语句)就可以描述.因此,即使应用程序使用解释执行,其执行速度也很快.
- 其次,多数无线传感网应用并没有苛刻的实时性要求,解释执行速度可满足应用需求.例如,在图 12 所示的多跳数据收集应用中,解释执行的速度并未影响应用运行的结果.

从无线传感网的现状来看,用户需要根据应用特点在网络正常运行开销和重编程开销之间权衡.从长远来看,随着无线传感网规模的扩大,传统的传输二进制镜像支持重编程的方法可能效率很低,甚至不能胜任重编程工作.因此,ReLog 选择了解释执行机制.

References:

[1] Yick J, Mukherjee B, Ghosal D. Wireless sensor network survey. Computer Networks, 2008,52(12):2292−2330. [doi: 10.1016/j.comnet.2008.04.002]

[2] Wang Q, Zhu Y, Cheng L. Reprogramming wireless sensor networks: Challenges and approaches. IEEE Network, 2006,20(3): 48−55. [doi: 10.1109/MNET.2006.1637932]

[3] Hui JW, Culler D. The dynamic behavior of a data dissemination protocol for network programming at scale. In: Proc. of the 2nd ACM Conf. on Embedded Networked Sensor Systems (SenSys). ACM Press, 2004. 81−94. [doi: 10.1145/1031495.1031506]

[4] Gay D, Levis P, von Behren R, Welsh M, Brewer E, Culler D. The nesc language: A holistic approach to networked embedded systems. In: Proc. of the ACM SIGPLAN 2003 Conf. on Programming Language Design and Implementation (PLDI). ACM Press, 2003. 1−11. [doi: 10.1145/781131.781133]

[5] Mottola L, Picco GP. Logical neighborhoods: A programming abstraction for wireless sensor networks. In: Proc. of the Distributed Computing in Sensor Systems. Berlin, Heidelberg: Springer-Verlag, 2006. 150−168. [doi : 10.1007/11776178_10]

[6] Miller JS, Dinda PA, Dick RP. Evaluating a BASIC approach to sensor network node programming. In: Proc. of the 7th ACM Conf. on Embedded Networked Sensor Systems (SenSys). ACM Press, 2009. 155−168. [doi: 10.1145/1644038.1644054]

[7] Hossain MS, Islam ABMAA, Kulkarni M, Raghunathan V. μSETL: A set based programming abstraction for wireless sensor networks. In: Proc. of the 10th Int'l Conf. on Information Processing in Sensor Networks (IPSN). IEEE, 2011. 354−365.

[8] Hill J, Szewczyk R, Woo A, Levis P, Madden S, Whitehouse C, Polastre J, Gay D, Sharp C, Welsh M, Brewer E, Culler D. Tinyos: An operating system for sensor networks. In: Weber W, ed. Proc. of the Ambient intelligence. Berlin, Heidelberg: Springer-Verlag, 2005. 115−148. [doi: 10.1007/3-540-27139-2_7]

[9] Milner R. A Calculus of Communicating Systems. New York: Springer-Verlag, 1982.

[10] Koshy J, Pandey R. Remote incremental linking for energy-efficient reprogramming of sensor networks. In: Proc. of the 2nd European Workshop on Wireless Sensor Networks. IEEE, 2005. 354−365.

[11] Titzer B, Lee D, Palsberg J. Avrora: Scalable sensor network simulation with precise timing. In: Proc. of the Int'l Conf. on Information Processing in Sensor Networks (IPSN). IEEE, 2005. 477−482. [doi: 10.1109/IPSN.2005.1440978]

[12] Madden SR, Franklin MJ, Hellerstein JM, Hong W. TinyDB: An acquisitional query processing system for sensor networks. ACM Trans. on Database Systems, 2005,30(1):122−173. [doi: 10.1145/1061318.1061322]

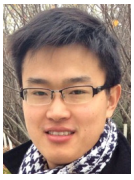
[13] Yao Y, Gehrke J. The cougar approach to in-network query processing in sensor networks. ACM Sigmod Record, 2002,31(3):9−18. [doi: 10.1145/601858.601861]

[14] Mottola L, Picco GP. Programming wireless sensor networks: Fundamental concepts and state of the art. ACM Computing Surveys (CSUR), 2011,43(3):19:1−19:51. [doi: 10.1145/1922649.1922656]

[15] Chu D, Popa L, Tavakoli A, Hellerstein JM, Levis P, Shenker S, Stoica I. The design and implementation of a declarative sensor network system. In: Proc. of the 5th ACM Conf. on Embedded Networked Sensor Systems (SenSys). ACM Press, 2007. 175−188. [doi: 10.1145/1322263.1322281]

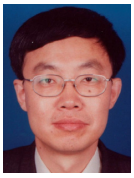
[16] Koshy J, Pandey R. VMstar synthesizing scalable runtime environments for sensor networks. In: Proc. of the 3rd ACM Conf. on Embedded Networked Sensor Systems (SenSys). ACM Press, 2005. 243−254. [doi: 10.1145/1098918.1098945]

朱晓瑞(1983－),男,江苏连云港人,博士生,主要研究领域为无线传感网应用编程与重编程.
E-mail: zxr@smail.nju.edu.cn



谢宏伟(1990－),男,硕士生,主要研究领域为无线传感网高效路由协议,室内定位.
E-mail: hongwei.xie.90@gmail.com

陶先平(1970－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件 Agent 技术,软件中间件技术,网构软件方法学,普适计算技术.
E-mail: txp@nju.edu.cn



吕建(1960－),男,博士,教授,博士生导师,CCF 高级会员,中国科学院院士,主要研究领域为软件自动化,面向对象语言与环境,并行程序的形式化方法.
E-mail: lj@nju.edu.cn