UC Davis
UC Davis Previously Published Works

Title
A Self-Adaptive Middleware for Efficient Routing in Distributed Sensor Networks

Permalink
https://escholarship.org/uc/item/4ff08304

Authors
Duan, Sisi
Sun, Jingtao

Publication Date
2015-10-09

Peer reviewed

eScholarship.org | Powered by the California Digital Library
University of California

A Self-Adaptive Middleware for Efficient Routing in Distributed Sensor Networks

Jingtao Sun
Architecture Science Research Division
National Institute of Informatics, NII
Tokyo, Japan
sun@nii.ac.jp

Sisi Duan
Department of Computer Science
University of California, Davis
CA, USA
sduan@ucdavis.edu

***Abstract*—Routing in sensor networks with unpredictable network connections and node failures is challenging due to the lack of knowledge about network dynamics in decision making. We present a self-adaptive middleware for efficient routing in distributed sensor network. At the heart of the proposed approach is the design of a policy-driven language to control the relocation of software components between sensor nodes. Accordingly, various changes in sensor networks can be dynamically adapted. Based on such a middleware, we provide a few approaches in building a practical routing protocol. For instance, nodes can adaptively switch their routing strategies according to the network stability or migrate some tasks to other idle nodes to prevent from node failures.**

***Index Terms*—Routing, Component-based, Relocation, Self-adaptation, Policy-driven language, Distributed sensor networks**

I. Introduction

Distributed sensor networks are widely used in a variety of industry applications, such as monitoring and control systems. However, the sensor nodes are usually constrained by limited storage and power, frequent failures, and mobility of nodes, resulting in unpredictable network connections, which makes routing less efficient. Existing routing approaches mainly fall into two categories: redundancy-based and knowledge-based. Redundancy-base routing [2], [15], [16], [18], [19] relies on the number of copies. Too little redundancy results in low delivery rate and too much redundancy results in high overhead. On the other hand, knowledge-based routing [5]–[7], [12], [17] requires nodes to obtain knowledge in order to make efficient routing decisions. However, knowledge about some network parameters does not necessarily improve the performance significantly.

Routing in sensor networks is challenging due to the lack of knowledge about network dynamics in decision making. When some sensors have high workload in collecting data, routing may create extra overheads, rendering the nodes unavailable. In addition, it is difficult to balance between redundancy and knowledge in a practical routing algorithm. Indeed, too many replicated messages may make nodes unavailable. It is also difficult to obtain useful network parameters due to the unpredictability of network dynamics.

Previous work [3], [4] proposed approaches to adapt to changes in distributed systems in architecture-level. For instance, Sun et al. [4] proposed a policy format to relocate software components between computers in component runtime system. In our work, we present a self-adaptive middleware to relocate software components between sensor nodes to adapt to various changes for efficient and practical routing. We also design and implement a specialized policy-driven language and a set of policies for adaptations. By using this language, the destination and conditions can be easily defined for relocation of software components. Also, the policies do not need to define the destination of components. Instead, our self-adaptive middleware will automatically decide for users. In addition, the user-defined policies can also be reused when the same condition happens. Based on our middleware, we provide several approaches in building an adaptive routing protocol. For instance, nodes can switch between different routing algorithms depending on the network connections. Furthermore, if too much storage or processing resource is consumed, a node can also relocate software components to other idle nodes so as to prevent from node and link failures. As a result, nodes can adaptively enjoy the benefits of both redundancy-based and knowledge-based protocols by not suffering from the drawbacks.

Our contribution can be summarized as follows:

- We present a self-adaptive middleware to adapt to various changes in distributed sensor network. We also design and implement a policy-driven language for adaptations.
- Based on the language, we define a set of policies and propose several approaches to build a self-adaptive routing protocol. For instance, when the network becomes unstable, nodes can switch to a more efficient routing algorithm to guarantee high delivery rate and low latency.
- With our developer-friendly middleware, developers can easily manage the policies for general purposes.

II. Approach

In this section, we discuss several scenarios in distributed sensor networks, based on which we introduce the system

978-1-4799-8697-2/15/$31.00 ©2015 IEEE

requirements and our proposed approach for building a self-adaptive middleware.

A. Scenarios

We consider two scenarios in distributed sensor networks. *Efficient routing strategy* guarantees that an effective and efficient routing strategy is used and *data sharing and distribution* effectively allocates node resources.

Efficient routing strategy. We consider two types of routing strategies: *proactive* routing, where routing information can be pre-computed and stored locally at nodes independent of traffic arrivals, and *reactive* routing, where routing information is computed in a on-demand manner when packets arrive. The former method is more efficient during routing. However, it requires the topology to be relatively stable and consumes computing resources even when no packets arrive. The latter method does not require any computing resources unless packets arrive. However, it might result in a longer latency. Our goal is to adaptively choose an appropriate strategy depending on the network stability and resource usage of nodes.

Data sharing and distribution. Apart from routing, sensor nodes are also constantly monitoring and collecting information depending on the applications. Therefore, it is desired to guarantee that software components will not consume too much computing power. Due to the fact that nodes in the same (physical) area are easily approachable, we can relocate the software components of a busy node to other idle nodes. Also, when a few nodes are collecting overlapping data, they can notify each other about the results, preventing busy nodes from consuming too much resources.

B. Requirement

To solve the above problems, we proposed a self-adaptive middleware system that meets the following requirements.

Self-adaptation. Existing distributed sensor networks usually employ a pre-determined network structure, e.g., client/server model, peer-to-peer model, and master-slave model, etc. However, the requirements of sensors/applications may often change and our distributed network constitution is desired to be self-adaptive to various changes.

Separation of concerns. Sensor network applications and adaptation modules should be defined independently. Our adaptation mechanism should be abstracted away from the underlying systems so that both the functions of software components and user-defined policies can be reused.

Dependability. Centralized management often becomes the bottleneck. Our middleware should to be built in a fully distributed environment and also guarantee data consistency.

General-purpose. Various applications are running on top of the sensors. The proposed approach should be implemented to support general-purpose applications. It should also be independent with other application-specific tasks.

Besides the above requirements, sensor nodes usually have limited processing and storage resources. Therefore, our system is desired to consume minimum resources for adaptability while most existing adaptation approaches explicitly or implicitly assume that their targets have enriched resources.

C. Approach

At the core of our approach are two key ideas. First, we develop a set of policies to relocate software components as a basic adaptation mechanism. Second, we design a language for specifying adaptation-policies. Based on such a design, our system can reuse both the software components and the policies in sensor networks.

Deployment software components on sensors. In general, an application consists of one or more software components. Our middleware system is dynamically adaptive to the changes through relocation of software components between sensor nodes, according to the deployment policies. When the software components are migrated to destination, the local and remote nodes can communicate with each other using our dynamic methods invocation mechanism. In addition, both the class file and the state of software components can be migrated so that our middleware can rapidly restart to handle interruptions.

Policy-driven language. To reduce the cost of programming software components and to provide precise definition of the software components, we design a policy-driven language for adaptations. Each software component can have one or more policies, where each policy is defined as a *conditions* and *actions* pair. The condition is written in a first-order predicate logic-like notation, where predicates reflect information about the system and application. An action represents the deployment and duplication of components for adaptation purposes, instead of application-specific behaviors such as communications and state transitions. In addition, our policy-driven language supports execution of multiple policies and conflict resolution mechanism.

III. POLICY-DRIVEN LANGUAGE

Our policy-driven language consists of four constructs, as shown in Fig. 1. Based on such a flexible language, different types of policies can be specified easily.

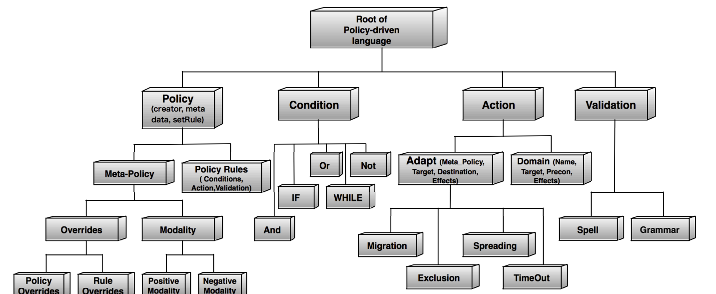

Fig. 1: Policy-driven Language.

A. Design of Policy-driven language

We use several notations to better describe our language, as shown in TABLE I. Our middleware enables users to specify

user-defined policies for adaptations by means of the expressions. Each expression contains *meta-policies*, *conditions for adaptation*, and *destinations of relocation*. The destinations of relocation is activated only if the conditions are true.

TABLE I: Notations.

Notation	Meaning
current	Current node
$\mathcal{L} = \{\ell_1, \ell_2, \ell_3, \cdots\}$	Location names
$\mathcal{X} = \{x_1, x_2, x_3, \cdots\}$	Location variable names
$\mathcal{S} = \{S_1, S_2, \cdots\}$	Meta data of policies
$\mathcal{C} = \{C_1, C_2, \cdots\}$	The identifiers of components with conditions
$\mathcal{N} = \{\mathcal{N}_1, \mathcal{N}_2, \cdots\}$	message names

Policy Expression. We define $\mathcal{D} = \{D, D_1, D_2\}$ as located process expressions, which is *the smallest set containing the following expressions*. In the expressions, C represents a set of conditions belonging to E, E_0 is often represented by E, σ represents all the execution of policies that are blocked, and τ is an action invoked as a callback function.

$$
\begin{array}{rll}
D,\, D_1,\, D_2 ::= & \ell[\mathrm{S} \mid \mathrm{E} \mid \mathrm{P}] & \text{(Located component)} \\
\mid & D_1 \parallel D_2 & \text{(Distributed component)} \\
S,\, S_1,\, S_2 ::= & S_1 > S_2 \parallel S_1 < S_2 & \text{(Meta policy)} \\
\mid & \sigma & \text{(Block execution)} \\
E,\, E_1,\, E_2 ::= & \mathrm{C}\ \textit{then}\ \mathrm{M}\ \textit{in}\ \mathrm{E} & \text{(Conditional action)} \\
\mid & E_1 + E_2 & \text{(Alternative selection)} \\
\mid & 0 & \text{(Termination)} \\
\mathrm{M} ::= & \mathrm{moveTo(x)} & \text{(Migration)} \\
\mid & \mathrm{copyTo(x)} & \text{(Duplication \& migration)} \\
\mid & \mathrm{remove(x)} & \text{(Elimination)} \\
\mid & \tau & \text{(Internal execution)} \\
P,\, P_1,\, P_2 ::= & P_1, P_2 & \text{(Composition)} \\
\mid & \mathrm{A} & \text{(Component)} \\
\mid & \epsilon & \text{(No component)}
\end{array}
$$

The policy expressions can be used to construct different policies. For instance, ℓ_1**[C then moveTo(**ℓ_2**)** | **A]** means that if condition C is true, a component A located at ℓ_1 is relocated to ℓ_2, where moveTo(ℓ_2) represents the migration to ℓ_2. Note that P in ℓ[C then E | P] intends to define application-specific processing independent of the policy, which is beyond the scope in the representation of the language.

In addition, ℓ**[**C_1 **then copyTo(**ℓ_3**) +** C_2 **then callback in remove** | **A, B]** means that if condition C_1 is true, two components A and B are copied and the copies are deployed at ℓ_2. Otherwise, if condition C_2 is true, the policy executes a callback function in A and B and then terminates A and B.

Policy Meta-Data. Our policy language supports execution of multiple policies, which may cause conflicts. Therefore, we imported meta-policy specifications for conflict resolution. A policy object includes rule-policy and meta-policy. Rule-policy is used to define the main body of user-defined rules. Meta-policy is used to specify which modality holds precedence over other policies, as described in details in §III-B.

Policy Condition. We define the policy conditional functions of our language as first-order logic predicates. The set of conditions C is the smallest set containing the following expressions: $C, C_1, C_2 := \phi \,|\, \neg C \,|\, C_1 \wedge C_2 \,|\, C_1 \vee C_2 \,|\, true \,|\, false$, where ϕ is a logical predicate symbol and returns either true or false with more than a zero parameter. Our policy-driven language supports the operators AND, OR, NOT, IF, and WHILE, based on which complex conditions can be built.

The current implementation provides several built-in functions as follows, where each predicate can have zero or more parameters and return true or false.

- **exist(A,**ℓ**) (exist :** $\mathcal{P} \times \mathcal{L} \rightarrow$ **true or false)** returns true if the same or compatible component(s) of component A exists at location ℓ, otherwise it returns false.
- **delay(time) (delay :** $\mathcal{T} \rightarrow$ **true or false)** blocks the subsequent executions for the time interval and then returns true, where $\mathcal{T}$ is an infinite set of relative time values.
- **received(N,** ℓ**, A) (requested :** $\mathcal{P} \times \mathcal{N} \times \mathcal{L} \rightarrow$ **true or false)** returns true if the component that the policy is assigned to receives a message labelled as N from component A.

User-defined functions are implemented inside components, meta policies, or the runtime system. By using these conditions, the policy actions can be triggered.

Policy Action. When the monitor of the system detects application or network changes, our pre-defined policies are immediately run to determine the destinations of software components in order to dynamically adapt to the changes. Since the language is designed for application developers, they do not need to define the destination of components. Instead, our middleware can automatically select destinations. The users only need to define the information of relocated components and the methods they intent to invoke.

We provide four major adaptation formats. 1) The *migration policy* is the basic adaptation format, which can be used to define the most basic adaptation policy as ℓ**[exist(A, another) then moveTo(another) in E** | **A]**. 2) The *spreading policy* is an extension of migration policy, where software components can be duplicated and the clones are relocated to the destination sensors or specified applications. We define it as **current[¬exist(A, another) then coypTo(another) in E** | **A]**. 3) The *exclusion policy* is also an extension of migration policy, which can be used when certain conditions are true. Since software components may be relocated repeatedly, the mutex functions is contained in node or application. In this case, our language can use this policy to allow software components to return back to its original node/application or transfer it to a safe node/application. It is defined as **current[exist(A, another) then moveTo(another) in E|A]**. 4) The *timeout policy* uses a user-defined timer and the software components can be dynamically changed using the timer. It is mainly used for the maintenance of systems and is defined as ℓ**[delay(**t**) then moveTo(another) in E** | **A]**.

Policy Validation. Our policy-driven language supports various well-known spell-checkers to check the translation in PO files. Currently there is only standalone support for Aspell. Spell-checking of one PO file or a collection of PO files can be performed directly by sieving them through one check-spell

(Aspell) or check-spell-ec sieves. The sieve will report each unknown word, possibly with a list of suggestions, and the location of the message (file and line/entry numbers). It can also be requested to show the full message, with unknown words in the translation highlighted. In addition, our language also support grammar checks. In the current implementation we use an open source grammar and style checker, called *LanguageTool*. It supports a number of languages to greater or smaller extent.

B. Policy Conflict resolution management

Due to the nature of sensor networks environment, we expect that several policies are applicable to every domain, which could lead to potential conflicts. For instance, when the software components of $node_1$ are required to be relocated to $node_2$ while the software components of $node_2$ are required to be migrated to $node_1$, resulting in a conflict. In order to resolve such conflicts, we assign each policy with a priority, e.g, using negative and positive. For simplicity, we define S= $S_1, S_2, \cdots$ to be a set of meta-policies, If $S_1 > S_2$, S_1 will be executed first. We use $S[S_1 > S_2$ then $S_1 \parallel S_1 < S_2$ then $S_2 \parallel \sigma \mid P]$ to denote such behavior, where P is the policy object defined by users and σ represents that both of S_1 and S_2 are blocked. In other words, if the negative and positive meta-policies encounter in same node, the permission of positive meta-policy is higher than negative meta-policy. However, if the policies have the same priorities in one node, both of them have to be blocked until a new policy can distinguish them or the administrator manually modify their priorities.

We use three kinds of meta rules:

- **metaRule(Policy, positive/negative)** Default precedence that can be set for a policy.
- **metaRuleAction(ConflictActions, positive/negative)** It can determine the order of execution for policies.
- **metaRuleBlock(ConflictActions)** Conflicting policies can be temporarily stopped until priorities are modified by other policy or system administrator.

By default, the meta rules associated with actions are given the highest priority than default meta rule without actions.

IV. Design and Implementation

The proposed self-adaptive system can dynamically deploys components to define application-specific functions at sensors according to user-defined policies. In this section, we introduce the design of our system.

A. System Architecture

Our self-adaptive system is built between the JVM (Java Virtual Machine) and distributed applications which are running on top of each sensor. Our self-adaptive system consists of three parts as shown in Fig. 2: *Adaptive Protocol Core*, *Components Runtime System*, and *Adaptation Manager*.

The *Adaptive Protocol Core* of our middleware provides basic parent class and several custom interfaces and tools for our system. The *Components Runtime System* is responsible for executing software components on sensors, migrating software components between sensors, and enabling them to invoke methods at the destination. The local sensor and destination sensors can communicate after migration according to naming inspection mechanism. The *Adaptation Manager* is responsible for interpreting user-defined policies. When user-defined conditions are different from the changes collected by system&resource&network monitor, software components will be relocated among runtime systems. When different policies are controlled by the same component, conflicts may occur. The conflict resolution management mechanism is used to determine the order of executing policies to resolve the conflicts. We also use a set of replicated destination databases to store the policies, where nodes can fetch the policies in a on-demand manner. It also supports automatic distribution of policies when new policies are added.

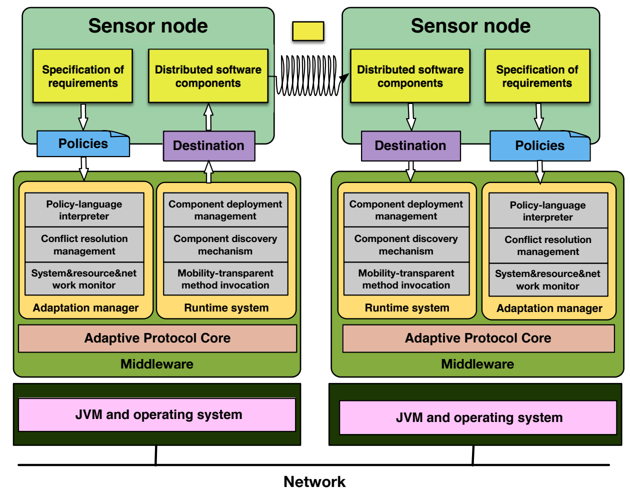

Fig. 2: System Module.

When external changes was notified by system&resource&network monitor, the policy-driven interpreter will invoke the user-defined policies according to the changes. The interpreter can then execute the policies and notify the runtime system the results. When the runtime system receives the results, the software components can be relocated to their destinations according to the policies. In this way, routing protocols can be dynamically changed from one to another.

B. Component runtime system

Every runtime system allows each component to have at most one activity through the Java thread library. When the life-cycle state of a component is changed, e,g creation, duplication, migration, and termination, the runtime system issues specific events to the software component. To capture such events, each component have more than one listener object. Each listener object implements an interface that hooks the events issued before or after changes in its life-cycle state. Through this method, we can easily hide the differences between the interfaces of objects at different nodes. In addition, different runtime systems can exchange components through TCP channels by using Object Input/Output Stream.

When a component is transferred over the network, both the code and the state are transmitted into a bit stream using Java's object serialization package and then transferred to the destination. The runtime system at destination side receives and unmarshals the bit stream. When components are deployed at destination, their methods can also be invoked from other nodes.

C. Adaptation mechanism

By default, each runtime system has its own adaptation manager. Each adaptation manager periodically advertises its address through UDP multicasting and nodes return their addresses and capabilities through a TCP channel. The adaptation manager also evaluates the network or application changes such as user requirements and resource availability. Each policy is specified based on the language defined in the policy expression, as described in §III-A. The adaptation manager offers an interpreter, which consists of three parts. The first part is responsible for defining meta information of policies, denoted by $S, S_1, S_2, \cdots$ to avoid conflicts. The second part is responsible for evaluating the conditions of adaptations, denoted by $C, C_1, C_2, \cdots$, as first-order logic predicates with predicates that reflect various system and network properties, e.g., the utility rates and processing capabilities of processors, network connections, and application-specific conditions. The third part is responsible for triggering actions of software components, denoted by $P, P_1, P_2, \cdots$, based on the operational semantics.

V. Routing

We provide a few approaches in building a practical and efficient routing protocol using the four policies shown in §III-A. We employ a two layer master-slave structure, as shown in Fig. 3. Each master node manages the information of a group of slave nodes. We assume slave nodes in the same group are physically or logically close such that each node can reach other nodes either directly or over multiple hops. In addition, when the master node fails, another slave node can be chosen to be the master node. Master nodes of different groups can also exchange information so that software components can be migrated across groups.

Change of routing strategy. Our design provides the flexibility to switch routing strategies at a node to address the problem described in §II-A. Our protocol switches between *proactive routing* and *reactive routing* depending on the network connectivity. When the network is relatively stable, we use proactive routing. Each node periodically pings its neighbors and reports to the master node if certain nodes are not responding. Master node of each group collects information from its slave nodes. When the number of new nodes and faulty nodes within certain period exceeds certain threshold, we consider the network not stable. It then notifies the nodes to switch to reactive routing. This approach provides accurate information for nodes to smartly choose an appropriate routing strategy. However, it requires the master node to maintain more information. An alternative approach is to switch between proactive routing and

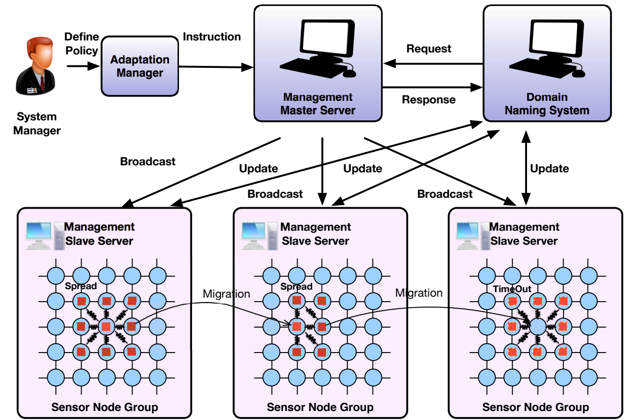

Fig. 3: Master-slave structure.

reactive routing periodically using timeout policy. Specifically, node 1 sets a timer and migrates its software component of routes computation to node 2. Before the software component is migrated back, node 1 uses reactive routing. In this approach, master node does not maintain information about network stability yet it may result in a less efficient routing strategy.

Routes computation sharing. In the above example, although nodes switch to reactive routing and routes do not have to be computed, we can still compute the routes so that the information can be directly used when the network becomes stable. Also, a node has limited resources and may not be able to compute the routes while processing other requests. In both scenarios, the node can migrate the tasks to other idle nodes. We model the route computation as a software component. The node can use either migration policy or spreading policy. To use the migration policy, the software component of routes computation at node 1 is simply migrated to node 2 and routes of node 1 are computed at node 2. In this case, node 1 simply contacts node 2 to obtain all the paths and uses during routing. On the other hand, to use the spreading policy, the software component is first cloned and then migrated to node 2. In this case, if node 1 cannot compute all the routes, it can contact node 2 to obtain the paths.

Data sharing. As discussed in §II-A, sensors constantly collect information and send to different nodes depending on the applications. When routing consumes too much resources, data collection component may become unavailable, or vice versa. We use spreading policy to handle such failure. For instance, we set a threshold for the storage usage. When the storage usage at node 1 exceeds the threshold, the spreading policy is run, i.e., node 1 copies the software component of data collection and migrates it to node 2. Note that node 2 must be in the same physical area and the storage usage is smaller than the threshold, i.e., it can also collect the same data as node 1. When node 2 collects the corresponding information, it simply shares the data with node 1. In this case, even when node 1 fails to collect certain data, it can still obtains from node 2.

Distributed software migration. In the above approaches, node 1 can contact the master node to obtain an available node for software components migration. It can also randomly picks a node 2 from its direct neighbors. In the latter case, node 2 may already have the same or conflicting software component. In this case, node 2 uses the exclusion policy until the software component reaches an available node 3. When the software component is deployed at node 3, node 3 sends a message to node 1 to notify the result.

VI. Related Work

Routing in distributed sensor network has been widely studied. Existing work fall into two main categories of routing strategies. The first type is replication-based, where a message is replicated and sent throughout different paths to increase the probability of the message getting delivered, e.g, flooding [2], [15], [16] and epidemic routing [18]. However, replication can be very expensive. The second type is knowledge-based, where different approaches are taken to optimize routing depending on different metrics [5]–[7], [12], [17]. The drawback is that there is no guarantee that physically close nodes are easy to communicate with low latency and it does not work well in mobile networks. In comparison, adaptive routing has been proposed [10], [14]. In Island Hopping [14], nodes are grouped into clusters where nodes can only communicate with nodes in the same cluster. It routes data through the mobility of nodes, i.e., using *data carrier*. Context-aware routing [10] manages different network attributes and nodes adaptively use a combination of different metrics to predict the node with the highest probability of delivery.

Several work built approaches to adapt to the changes in distributed systems and mobile applications through relocation of software components between different computers/servers. Sun et al. [4] proposed a policy format to define requirements of users on a self-adaptive middleware that relocates software components between computers in architecture-level. Different from previous work, we propose a specialized policy-driven language for adaptations in distributed sensor networks. We specialize the destination of relocated components using our language. Ioannis et al. [3] present a connection-based architecture for self-organizing software in distributed systems. Like other architecture-level adaptations, they intended to customize their systems by changing the connections between components instead of the internal behaviors inside them. Weyns et al. [1] proposed a mobile storytelling application that employs a social recommender using ActivFORMS. They added a self-adaptive layer on top of the application to adapt to changes. They also underpin the need for an integrated verification approach for self-adaptive systems that combines offline and online verification. Different from our work, they do not raise the adaptation conditions to the level of language so that they can not be widely applied. Similar with our work, several previous research [8], [9], [13] also proposed language-based adaptation for self-adaptive systems. However, they do not define the destinations of software components during relocation. Therefore, it is difficult for system administrators to monitor the migration trajectory. When the system fails, it is difficult to fix the failures in a timely manner.

VII. Conclusion and Future Work

In this paper, we present the design of a self-adaptive middleware in distributed sensor networks. Based on such a middleware, we provide several approaches in building a practical and efficient routing protocol, e.g., nodes can smartly adjust their routing strategies according to the network connectivity and also migrate or spread their tasks (modeled as software components) to other nodes to prevent from node failures and message loss. Our developer-friendly middleware also makes it easy to add and manage policies for general purposes in sensor networks. In the future, we will further develop, implement, and evaluate the routing protocol using our self-adaptive middleware. We also look forward to handling more scenarios in sensor network routing to build a fully-fledged protocol.

References

[1] D. Weyns, S. Shevtsov, S. Pllana: *Providing Assurances for Self-Adaptation in a Mobile Digital Storytelling Application Using ActivFORMS*, SASO, pp. 110–119 (2014).

[2] M. Grossglauser, and D. N. C. Tse: *Mobility increases the capacity of ad hoc wireless networks*, IEEE Trans. Netw. 10(4), pp. 477–486 (2002).

[3] I. Georgiadis, J. Magee, and J. Kramer: *Self-Organising Software Architectures for Distributed Systems*, WOSS, pp. 33–38 (2002).

[4] J. Sun and I. Satoh: *Dynamic Deployment of Software Components for Self-Adaptive Distributed Systems*, IDCS, pp. 149–203 (2015).

[5] E. P. C Jones, L. Li, J.K. Schmidtke, and P. A. S. Ward: *Practical routing in delay-tolerant networks*, IEEE Trans. Mob. Comput. 6(8), pp. 943–959 (2007).

[6] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein: *Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet*, ASPLOS-X, pp. 96–107(2002).

[7] B. Karp and H. T. Kung: *GPSR: greedy perimeter stateless routing for wireless networks*, MobiCom, pp. 243–254 (2000).

[8] L. Kagal, T. Finin, and A. Joshi: *A Policy Language for a Pervasive Computing Environment*, POLICY, pp. 63–74 (2003).

[9] M. Luckey and G. Engels: *High-quality specification of self-adaptive software systems*, SEAMS, pp. 143–152 (2013).

[10] M. Musolesi and C. Mascolo: *CAR: Context-aware adaptive routing for delay-tolerant mobile networks*, IEEE Trans. Mob. Comput. 8(2), pp. 246–260 (2009).

[11] A. Nayebi, H. Sarbazi-Azad, and G. Karlsson: *Routing, data gathering, and neighbor discovery in delay-tolerant wireless sensor networks*, IPDPS, pp. 1–6 (2009).

[12] T. S. E. Ng and H. Zhang: *Predicting internet network distance with coordinates-based approaches*, INFOCOM, pp. 170–179 (2002).

[13] D. Nicodemos, D. Naranker, L. Emil, S. Morris: *The ponder policy specification language*, POLICY, pp. 18–39 (1995).

[14] N. Sarafijanovic-Djukic, M. Piórkowski, M. Grossglauser: *Island Hopping: Efficient mobility-assisted forwarding in partitioned networks*, Technical Report, Duke University (2000).

[15] R. C. Shah, S. Roy, S. Jain, and W. Brunette: *Data mules: modeling a three-tier architecture for sparse sensor networks*, Ad Hoc Networks 1(2–3), pp. 215–233 (2003).

[16] T. Small and Z. J. Haas: *Resource and performance tradeoffs in delay-tolerant wireless networks*, WDTN, pp. 260–267 (2005).

[17] T. Spyropoulos, K. Psounis, and C. S. Raghavendra: *Single-copy routing in intermittently connected mobile networks*, SECON, pp. 235–244 (2004).

[18] A. Vahdat and D. Becker: *Epidemic routing for partially connected ad hoc networks*, SECON, pp. 226–235 (2006).

[19] Y. Wang, S. Jain, M. Martonosi, and K. Fall: *Erasure-Coding Based Routing for Opportunistic Networks*, WDTN, pp. 229–236 (2005).