Dynamic Sets – A Programming Abstraction for Ubiquitous Computing and the Internet of Things

Matthias Prellwitz, Helge Parzyjegla, Gero Mühl, Dirk Timmermann
University of Rostock, Germany
{matthias.prellwitz,helge.parzyjegla,gero.muehl,dirk.timmermann}@uni-rostock.de

ABSTRACT

Applications in the areas of Ubiquitous Computing and the Internet of Things are often facing the problem that at design time it is not known what devices will be available at runtime and what preferences the user will have. Due to this gap between design time and runtime, mechanisms are needed to postpone decisions until the application is deployed and to adapt these decisions while it is running. Unfortunately, such mechanisms are currently mostly missing.

In this paper, we propose *dynamic sets* as a programming abstraction that allows an application originally written for using a single device to interact with a set of devices of the same type instead. Using a dynamic set, an application can transparently call a method on a set of devices using a local proxy of the set that implements the very same interface as a device the application was originally written to interact with. Applications that are aware of dynamic sets can change the criteria used to select the members of a set or derive a new set from existing sets. Besides invocation mechanisms, dynamic sets provide functionality for result aggregation, automatic member management, update notifications, and derived business methods. Finally, applications can be exported making it possible to combine existing applications to flexible and dynamic mashups.

CCS Concepts

•**Software and its engineering** → *Middleware;* **Abstraction, modeling and modularity;**

Keywords

Programming abstractions, object grouping.

1. INTRODUCTION

Applications in the areas of *Ubiquitous Computing (UbiComp)* and the *Internet of Things (IoT)* often rely on dynamic device ensembles with a potentially large and varying number of devices that interact with users and that cooperate with each other by exchanging sensor data and executing actor commands. This trend is driven by continuously decreasing hardware prices of sensors, actuators, and mobile devices (e.g., smartphones) bringing scenarios and corresponding applications to the mass market. However, with increasing opportunities and complexity, developing and maintaining systems leveraging dynamic device ensembles becomes a difficult challenge. Since the number and characteristics of the available devices as well as the user preferences may change over time, it is not feasible to make all decisions at design time. Therefore, mechanisms are needed that allow to postpone as much of these decisions as possible until applications are deployed and to adapt these decisions based on the devices that are available to applications at runtime. Unfortunately, such mechanisms are currently missing to a great extent.

In this paper, we present a programming abstraction called *dynamic sets* that enables an application to interact with a varying number of devices in the same way it interacts with a single device. To achieve this, an interface for a set of devices is automatically generated from the interface for a single device of the respective type. The generated interface has the same methods with the same signatures as the original interface and a local proxy object enables the application to issue method calls and to receive result values in return. The default semantics is that a call to a method of a set is sent to all individual devices currently in the set and that the results coming back from the devices are aggregated to a single result using one of the available result aggregation policies. This way, it is also possible to hide the interaction paradigm (e.g., pull via RMI or push using publish/ subscribe) and the underlying transport protocols used to access services on remote devices (e.g., RMI, REST, CoAP) from the applications. Additionally, it is possible to expose the interface of a set or of an application to other applications. This allows application programmers and deployers to realize mashups facilitating flexibility and extendability.

The remainder of the paper is structured as follows: The basic idea of dynamic sets and how applications can transparently use their default semantics is presented in Sect. 2. Section 3 discusses the additional functionality that an application can use if is aware of the fact that it is interacting with a dynamic set. Section 4 introduces advanced concepts enabling the programmer to customize the functionality of a dynamic set and Sect. 5 discusses flexible mashups, an optimization switching between push and pull, and quality of service aspects. Related work is discussed in Sect. 6 and our conclusions and future work are presented in Sect. 7.

MOTA '16, December 12-16 2016, Trento, Italy
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4669-6/16/12. . . $15.00
DOI: http://dx.doi.org/10.1145/3007203.3007213

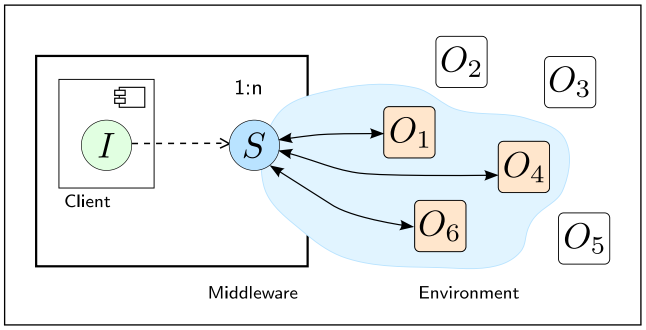

Figure 1: Basic scenario.

```
interface ILight {
   boolean isPowered();
   void    setPowered(boolean pow);
   boolean isDimmable();
   int     getRoom();  ...
}
```

Listing 1: Plain business interface.

2. DYNAMIC SETS

We introduced dynamic sets in a previous work [8] as a programming abstraction that allows applications to interact with a variable number of remote objects in the same way they usually interact with a single object of that type. Therefore, it is fully transparent by default for the application that is interacting with a set of objects and not with a single object. Fig. 1 sketches the basic scenario of dynamic sets. The application I calls a method at the local proxy S that represents the current set of objects and that has exactly the same business interface (i.e., the same methods and method signatures) as a single object of that type. The proxy determines the current members of the dynamic set (i.e., O_1, O_4, and O_6), replicates the method calls to all members, and aggregates the three return values to a single return value that is passed back to the calling application.

Listing 1 shows the business interface `ILight` with some methods. This interface represents an electric light and is used for our example application. For instance, the method `setPowered` can be used to power a light either on or off. Per default when calling a method on the local proxy of a dynamic set of type `ILight`, the method is called on all objects that are currently members of the set.

2.1 Declaration and Instantiation

An instance of a dynamic set is declared at design time by the application programmer. She or he annotates the application code and puts the annotation `@DynamicSet` in front of a class member. To become a set member, objects must

```
class MyLightApp {
  @DynamicSet
  @SelectionConstraint(
      "powered = true AND room = '268'")
  @Aggregation(
      clazz = AllAggregation.class,
      method = "isPowered, isDimmable")
  ILight light;
}
```

Listing 2: Exemplary dynamic set declaration.

```
class AllAggregation<T> {
  @OnAggregate
  boolean allValuesTrue(Object[] o) {...}
}
```

Listing 3: Exemplary aggregation method.

implement the interface of the class member, which is tested automatically. Additionally, *selection criteria*, which must be satisfied by an object to become a member of the set, can declaratively be specified along with the set definition using the `@SelectionConstraint` annotation. The selection criteria consists of constraints that evaluate attributes of an object or return values of corresponding getter methods. A dynamic set, that has been declared at design time, is created by the middleware runtime environment and supplied to the application by using dependency injection. A dynamic proxy for the dynamic set is generated automatically from the respective object interface using Java's reflection mechanism. According to the default semantics of dynamic sets, calling a method on the local proxy triggers a corresponding method call on all members currently in the set.

Listing 2 shows how the code of the example application is annotated such that a dynamic set (whose members are of type `ILight`) is injected into the class member `light`. This happens when the instance of `MyLightApp` is created by the runtime environment. The given selection criteria ensures that only lights that are located in the specified room and that are powered on become members of the set. After the injection, the application can call methods on the set proxy.

2.2 Configuration

By choosing options and policies, dynamic sets can be configured such that their semantics deviates from the default semantics. Configuration can be done by the application developer at design time, by the local administrator at deployment time, and by the actual user at runtime. The resulting semantics of the dynamic set, then, emerges from the combined configurations, where a later configuration affecting a certain aspect usually overwrites an earlier configuration affecting the same aspect.

At design time, the application developer can declare dynamic sets using code annotations. She or he usually also specifies the application-specific default semantics of the dynamic set overwriting or complementing the predefined fallback default semantics. At deployment time, middleware maintainers can overwrite or complement the semantics with deployment-specific configurations options or policies. This is achieved by providing configuration files, such as a *deployment descriptor*, together with the application to be deployed. At runtime, configuration can be adapted using the configuration API of dynamic sets. This API can, for example, be used to implement runtime optimizations or to enable the user to express her or his personal preferences.

2.3 Return Value Aggregation

Since the signatures of a method of a dynamic set and of the same method of a corresponding individual object are identical, it is necessary to aggregate the return values resulting from calls to the former method to a single return value if more than one call to the latter method has been issued. To achieve this, the programmer can choose from a number of predefined aggregation methods (e.g., minimum,

```
interface DynSet<T> extends Set {
  // derivation from existing sets
  DynSet  subset(Constraint ... c);
  DynSet  union(DynSet... s);
  DynSet  intersection(DynSet... s);
  DynSet  complement(DynSet a, DynSet b);
  DynSet  difference(DynSet a, DynSet b);
  // selection constraints
  void    select();
  void    setConstraints(Constraint... c);
  Constraint[] getConstraints();
  // member updating
  void    setAutoUpdate(boolean b);
  boolean isAutoUpdate();
  // set listener
  void    addListener(Object listener);
  void    removeListener(Object listener);
}
```

Listing 4: Set management interface.

maximum, average, majority, first). Depending on the semantics of the aggregation method, a different number or portion of the result values must be available before the single return value can be determined and passed back to the application. For example, returning the maximum requires all return values to be available, while for returning the first value, a single return value is sufficient. Besides using predefined aggregation methods, it is also possible for the developers to implement new aggregation methods.

Listing 3 shows an application-specific aggregation using a separate class. The annotation `@OnAggregate` labels the `allValuesTrue` method as aggregation method so that an array of values is passed and that returns a single value. The use of this new aggregation method is declared in Listing 2 using the `@Aggregation` annotation. Here, this annotation specifies that the new aggregation method is implemented in the class `AllAggregation` and that it should be applied to the `isPowered` and `isDimmable` methods.

3. SET MANAGEMENT

The functionality of dynamic sets described so far is fully transparent to the application. In particular, the developer does not need to adapt the application code, because every application that is able to deal with a single object can also deal with a dynamic set. The deployer of an application, however, can configure a dynamic set (e.g., provide a specific aggregation method via a deployment descriptor) such that it better fits the intended application functionality.

If the application itself is aware of the fact that it is interacting with a dynamic set, additional possibilities arise and the application can call methods of a dynamic set not contained in the original business interface. This means that the middleware enriches the set interface with methods that can, for example, be used to derive new sets from existing sets, to iterate over the set members, to request current set statistics, or to change the selection criteria of a set. Next, we discuss those methods that allow to manage a dynamic set. We first describe the manual management interface and, then, the automatic management interface.

3.1 Manual Management

The manual management interface is used by the programmer after the set has been created and injected by the

```
light.addListener(new Object(){
  @OnMemberAdded
  void foo(Object ... o) { ... };
  @OnMemberRemoved
  void bar(Object ... o) { ... };
});
```

Listing 5: Set listener implementation.

runtime environment. Listing 4 depicts the corresponding interface `DynSet` that extends the `Set` interface of Java's Collection API. The programmer can use the inherited methods to iterate over the individual members of the set (`iterator`), to get the current number of set members (`size`), to test whether an object is currently a set member (`contains`), and to add and remove objects from the set. The first method block allows to derive a new dynamic set from existing sets. A *subset* can be created by adding further constraints to an existing set and with methods corresponding to the usual set operations, it is possible to create the union, intersection, or relative complement of two (or more) sets. The second block allows to set the select criteria of the dynamic set via the `setConstraints` method. This triggers the middleware to evaluate the new criteria and to update the set accordingly such that after the call has returned, exactly those objects are a member of the set that satisfy the new criteria. To trigger the middleware to reevaluate the current select criteria without changing it, the `select` method can be called.

3.2 Automatic Management

The automatic management interface is also depicted in Listing 4. The methods of the next to last method block can be used to turn on or off a functionality called *auto update* using the `setAutoUpdate` method. When it is turned on, the middleware autonomously reevaluates the selection criteria of the dynamic set in order to update the set members periodically as well as calling several methods of the manual management interface raises a runtime exception. This is to avoid unintended interference between manual method calls and those triggered by the auto update functionality. To let the application react on specified events, it is possible to add listeners to a dynamic set using the `addListener` method, while existing listeners can be removed with the `removeListener` method.

Listing 5 depicts a code snippet that shows how listeners can be added using the `addListener` method. In this example, the methods `foo` and `bar` are marked with the annotations `@OnMemberAdded` and `@OnMemberRemoved` to get the added or removed set members as parameters, respectively. As an alternative, it is also possible to annotate application methods of the object having a dynamic set as member (e.g., our example application class) with these two annotations. In case an object has more than a single dynamic set as member, the target dynamic set must be given as a parameter to the annotation, e.g., `@OnMemberAdded("light")`.

4. CUSTOMIZATION

The previous sections discussed how to transparently execute business methods on all set members and how to manage dynamic sets. On the one hand, this significantly eases controlling the devices represented by a dynamic set. On the other hand, it also restricts usable features to those of a single device wasting the potential of the many. For instance,

```
 1 interface CustomLights
 2     extends ILight, DynSet<ILight> {
 3   // remix of existing functions
 4   @Aggregation(
 5     clazz=Average.class,
 6     method="isPowered" )
 7   float getBrightness();
 8   // mixin of additional functions
 9   @Mixin(StepSwitch.class)
10   @Config(name="steps", value="10")
11   StepSwitch stepswitch();
12   default void brighter() {
13     stepswitch().forMore(
14       Invoke.method("setPowered")
15             .args(true) );
16     stepswitch().more();
17 } }
```

Listing 6: Custom interface with enriched business methods.

even if no lamp in a dynamic set is dimmable, the overall brightness might be adjusted by switching on a smaller or a larger fraction of the lamps. This way, applications gain an added value that goes beyond the features and, thus, the interface of a single device. To leverage this potential, we enable developers to declare custom business methods combining business logic with set semantics.

For this purpose, the developer can define a new interface as depicted in Listing 6 for our exemplary lighting application. The interface `CustomLights` extends the business interface `ILight` and the management interface `DynSet<ILight>` for dynamic sets (lines 1-2). Hence, a corresponding proxy generated and injected by the middleware can, as before, be used as simple lamp as well as managed as a dynamic set of lamps. For the latter, however, it does not require an additional typecast anymore. In addition, the developer can declare further business methods that are also implemented by the generated proxy later. The actual implementation logic to be used, however, is provided by either remixing existing business methods and set functions (e.g., different result aggregations) or by delegating method invocations to supplementary components containing the necessary code. These two options are described in the following.

4.1 Remix of Set Logic and Business Logic

The aggregation of return values to a single data value is a prerequisite to transparently invoke a method on a set of objects as if the method was called on a singleton object. As a direct consequence, the aggregated value needs to be of the same type as original return value. Moreover, the aggregated value has to make sense in the application context in which the method is called without surprising the developer. Often, this limits applicable aggregation functions to a few choices such as first, majority, or minimum and maximum. For business methods newly defined in the extended interface, however, we are not bound to these limits. Thus, we can combine and mix original business methods with different aggregation functions in order to compute and query additional data and information about the dynamic set.

Assuming that the dynamic set in our lighting application consists of non-dimmable lamps, we may define the overall brightness as the fraction of lights that are switched on. Correspondingly, we declare the method `getBrightness` in Listing 6 (line 7) to query the current brightness value. To actually compute the value, the method's `@Aggregation` annotation (lines 4-6) instructs the middleware to average (i.e., use `Average.class`) the results of the `isPowered` method. This method is provided by the original interface of the lamps and determines whether a lamp is switched on or off. Invoking the method on all set members and calculating the average (i.e., `true` interpreted as 1.0 and `false` as 0.0) finally gives the fraction of lighted lamps in the set.

4.2 Flexible Mixin of Advanced Features

Not all newly declared methods can be simply implemented by relaying the invocation to an already existing business method of the device and using a different aggregation function for obtained return values in order to compute desired results. Often, advanced features also require very specific and additional implementation logic. Besides writing own implementations, supplementing components and libraries may provide necessary logic and missing functions in a reusable fashion that can flexibly be mixed in own code. While several programming languages such as Python or Ruby natively support the concept of mixins, this design pattern has to be emulated in other languages. For our dynamic set implementation, we can realize mixins in Java by exploiting getter-based dependency injections in combination with default methods that are available in Java 8. Listing 6 (lines 9-16) gives an example.

The primary idea is to realize a stepping switch for our lighting application that gradually turns on the lamps of the dynamic set instead of switching them on all at once. For this purpose, the members of the dynamic set are subdivided in subsets of equal size. With each subsequent call, the switch turns on the lamps of a further subset increasing the overall brightness. As such a gradual invocation strategy is also sensible in other application contexts, we have factored out the implementation into an own `StepSwitch` class. The `@Mixin` annotation (line 9) instructs the middleware to also instantiate a new `StepSwitch` object whenever a dynamic set of `CustomLights` is created. Name/value pairs provided by optional `@Config` annotations (line 10) are passed to the instantiated object and can be used for configuration purposes, for example, to set the number of switching steps to 10 until the lamps of the whole set are turned on. Whenever the `stepswitch` method (line 11) is invoked later, this `StepSwitch` object is returned.

To leverage the `StepSwitch` functions, we build on the usage of default methods. Default methods enable developers to add functionality to interfaces by having method declarations being accompanied with a default implementation that is automatically inherited. This way, we can conveniently forward a call of the `brighter` method (line 12) for turning on more lights in the dynamic set to the appropriate functions of the `StepSwitch`. Therefore, we first pass a compiled invocation handler (lines 13-15) that specifies method and arguments to be called on a single lamp (i.e., `setPowered(true)`) in order to turn it on. Afterwards, we let the `StepSwitch` issue the call on a subset of lamps (line 16). Similarly, we can also define and implement a `darker` method that gradually switches off the lamps again.

5. MIDDLEWARE FEATURES

The following subsections describe middleware features of our dynamic set framework that improve the benefits and efficiency of using dynamic sets. We start by sketching how dynamic sets can be used to build flexible mashups and continue with introducing an optimization mechanism and discussing important quality of service aspects.

5.1 Flexible and Efficient Mashups

With dynamic sets, a programmer can implement applications utilizing an unknown number of devices and objects in a convenient way. In order to enable a programmer to reuse existing applications and to combine them to flexible mashups, an application can define a business interface and our middleware can be instructed to make this interface and, thus, the application available remotely via mechanisms such as Java RMI, REST, SOAP, and Apache Thrift. Similarly, our middleware can use different protocols to access the objects that are bundled into a dynamic set. To facilitate flexibility and extendability, an application is usually neither aware of the protocol the middleware uses to access the objects in a dynamic set nor of the protocol(s) it uses to export the functionality of the respective application. To achieve interoperability and integrate legacy technology, our middleware platform also utilizes gateways [2] that hide protocol heterogeneity and adopt transport bindings. Moreover special attention is paid to meet the requirements of resource constraints devices. This includes, for example, a lightweight SOAP over CoAP transport binding using an EXI-based XML compression [5] to make web services applicable in wireless sensor networks.

5.2 Adaptive Push/Pull

Frequent method calls on a dynamic set or on individual objects of a dynamic set may raise a problem that is already well known from remote method invocation implementations such as Java RMI and that is aggregated in case of dynamic sets by the fact that a method call may trigger a call on a potentially very large number of remote objects: since the programmer is not aware that the used objects are remote, she or he uses the objects in the same way as local objects. However, issuing too many remote method calls may waste resources and induce network congestion.

We have tackled this problem for calls to getter methods of an object by implementing an adaptive algorithm that switches between pull-based and push-based communication to optimize resource consumption. The algorithm considers both the *access frequency* of the application and the *update frequency* of the remote object to make its decisions. For example, if the access frequency of the application is large compared to the update frequency of the object, push-based communication is used. This means that the object pushes a change to the local proxy of the application which caches the latest values and simply returns this value when the application calls a getter method. Similarly, when the update frequency of the object is large compared to the access frequency, pull-based communication is preferred. This means that calling a getter method on an object results in a remote call. This optimization can not only be applied to getter methods, but also to other methods that do not change the object's state. Our preliminary evaluation results show that our adaptive algorithm reduces the resource consumption significantly in certain scenarios, while it usually does not have a negative impact on performance.

5.3 Quality of Service

Our dynamic set framework supports a variety of *Quality of Service (QoS)* options that can be configured by the developer, the deployer, or the user of an application. The supported options cover aspects such as error handling and retry strategies as well as strategies to limit latency or maximal value age. As another example, the local proxy of a dynamic set may be instructed to apply sampling or overbooking. *Sampling* is especially useful for large sets, because in that case it can make sense for applications that are satisfied with an approximated result to only query a certain number or fraction of the objects –usually chosen randomly– instead of asking all objects. *Overbooking* can be useful when a method shall be called on a certain number or fraction of objects known to be *unreliable*. Here, a call is distributed to a larger number or fraction of objects than actually necessary in order to avoid the additional latency introduced when the number of answering objects is too low and other objects must be queried in additional request rounds to get a result with the required precision. Another interesting aspect is the *spawn and call strategy*. Here, the local proxy object may, for example, be instructed to issue the method calls synchronously one after the other or to issue several asynchronous calls (up to a certain limit) simultaneously.

6. RELATED WORK

In the literature, much research on programming abstractions and, in particular, on transparently grouping objects exists. In the following, we discuss those approaches that are closely related to our approach.

In the area of sensor networks, several approaches supporting the idea of sensor fusion exist that have similarities with our approach. However, most approaches in this area mingle code for middleware functionality and for application logic in order to produce compact implementations with small footprints. For example, Aberer et al. [1] propose the *Global Sensor Networks (GSN)* middleware whose central concept is the *virtual sensor* abstraction which enables to declaratively specify XML-based deployment descriptors in combination with the possibility to integrate sensor data through plain SQL queries over local and remote sensor data. Independently, Kabadayi et al. [9] also proposed virtual sensors. Here, a virtual sensor is a software sensor combining readings from multiple physical sensors (e.g., to improve quality) that can be specified declaratively.

In the area of context-aware applications, Sehic et al. [11] propose a programming model for large-scale context-aware applications called Origins Model. In this model, an origin is the elementary application component and abstracts a single context source. It can, for example, wrap an original data source or filter, aggregate, compose, and infer data using data provided by other origins that are declaratively specified with selection criteria. Based on this selection, the data can be requested from the selected origins either synchronously by a normal call or asynchronously by using a future. In the latter case, completion handler can be defined that is called when the requested data is available. Using futures enables *promise pipelining* that can reduce latency by allowing to define a processing operation on top of a future instead of an actual value.

The following four approaches are located in the area of distributed system. These approaches and our work have in common that they are based on the same idea, which is to provide transparent access to an object group. Eugster et al. [3] present a programming abstraction called *Distributed Asynchronous Collection (DAC)* that groups a set of event objects to a collection. A collection is accessed via a local proxy that hides distribution and provides transparent access. The interface of DACs primarily focus on managing the members of the collection. In addition to pull-based methods (e.g., `add` and `contains`) with a single return value, DACs offer push-based methods that allow a client to register a callback to be notified in the future, e.g., when a new object is added to the collection. These methods are mapped to the underlying topic-based publish/subscribe paradigm. The main difference between DACs and dynamic sets is that DACs focus on delivering event objects representing state changes of objects (either using pull or push) to applications, while dynamic sets focus on enabling applications to transparently issue method calls (either synchronously or asynchronously) to a set of remote objects. Another difference is that DACs rely on publish/subscribe, while dynamic sets transparently support different transport protocols and interactions paradigms.

Felber et al. [4] describe an *Object Group Service (OGS)* that provides for fault tolerance and high availability by transparent object replication. The service enables an application to call methods of an object group via a local proxy in a completely transparent way, i.e., the application cannot distinguish between an object group and a singleton object that implements the same interface. Because of this transparency, applications can be made fault-tolerant without having to change their code. To achieve fault-tolerance for benign faults, it is sufficient to return any of the return values. Thus, a call can return when the first reply arrives, but it can be configured that the call returns after the majority of replies or all replies have arrived.

Fault-Tolerant CORBA (FT-CORBA) (cf. [6], chapter 23) and more recent approaches in the area of web services [10] also use transparent object replication. However, dynamic sets are not limited to object replication, since they can bundle arbitrary objects implementing the same interface and they also provide advanced functionality in addition to invocation mechanisms.

Picco et al. [7] propose *Distributed Abstract Data Types (DADTs)* that logically encapsulate a set of instances of an abstract data type in a distributed system. An application programmer can define an interface for a DADT (whose methods operate on the set) and she or he can restrict the scope of a DADT by defining a view representing a subset in a declarative way. Methods can transparently be invoked, for example, on a single instance, on all instances, or a declarative selector can be used to determine the target instances. It is also possible to iterate over the members and to access individual members.

7. CONCLUSIONS

In this paper, we discussed dynamic sets as a programming abstraction that eases the development of ubiquitous computing applications by bridging the gap between design time and runtime. With dynamic sets, an application that was originally written for using a single device can instead transparently interact with a set of devices of the same type. Moreover, an application that is aware of the fact that it is interacting with a dynamic set can make use of the advanced features of dynamic sets presented in this paper. These features include result aggregation, automatic management of the dynamic set, update notifications, and derived business methods. Aware applications can also change the selection criteria of the set according to their needs or derive a new set from existing sets. Finally, the possibility to define and to export application interfaces enables to combine existing applications to flexible and dynamic mashups.

For future work, we plan to further extend the functionality of dynamic sets. In particular, we want to implement means for an application to register listeners that are called if a member object enters a specified state or exhibits a specified state change. This enables an application to solely receive interesting update notifications instead of potentially a lot of useless updates. Furthermore, we plan to extend dynamic sets to support transactions and security mechanisms.

8. REFERENCES

[1] K. Aberer, M. Hauswirth, and A. Salehi. A Middleware for Fast and Flexible Sensor Network Deployment. In *32nd Int'l Conf. on Very Large Data Bases*, pages 1199–1202. VLDB Endowment, 2006.

[2] V. Altmann, B. Butzin, R. Balla, F. Golatowski, and D. Timmermann. A BACnet gateway for embedded Web services. In *2015 IEEE 20th Conf. on Emerging Technologies Factory Automation*, 2015.

[3] P. Eugster, R. Guerraoui, and J. Sventek. Distributed Asynchronous Collections: Abstractions for Publish/Subscribe Interaction. In *14th European Conference on Object-Oriented Programming*, number 1850 in LNCS, pages 252 – 276. Springer, 2000.

[4] P. Felber, R. Guerraoui, and A. Schiper. Replication of CORBA objects. In *Advances in Distributed Systems, Advanced Distributed Computing: From Algorithms to Systems*, pages 254–276. Springer, 1999.

[5] G. Moritz, F. Golatowski, and D. Timmermann. A Lightweight SOAP over CoAP Transport Binding for Resource Constraint Networks. In *8th Int'l Conference on Mobile Ad-Hoc and Sensor Systems*. IEEE, 2011.

[6] OMG. *The Common Object Request Broker: Architecture and Specification, Version 3.0.3.* Object Management Group, Inc., Mar. 2004. http://www.omg.org/spec/CORBA/3.0.3/.

[7] G. P. Picco, M. Migliavacca, A. L. Murphy, and G.-C. Roman. Distributed Abstract Data Types. In *On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *LNCS*, pages 1594–1612. Springer, 2006.

[8] M. Prellwitz, H. Parzyjegla, and G. Mühl. Dynamic Sets: A Programming Abstraction for Object Bundling. In *14th Int'l Workshop on Adaptive and Reflective Middleware*, pages 9:1–9:3. ACM, 2015.

[9] V. Rajamani, S. Kabadayi, and C. Julien. An Interrelational Grouping Abstraction for Heterogeneous Sensors. *Transactions on Sensor Networks*, 5(3):27:1–27:31, 2009.

[10] J. Salas, F. Perez-Sorrosal, M. Patiño Martínez, and R. Jiménez-Peris. WS-Replication: A Framework for Highly Available Web Services. In *15th Int'l Conf. on World Wide Web*, pages 357–366. ACM, 2006.

[11] S. Sehic, F. Li, S. Nastic, and S. Dustdar. A Programming Model for Context-aware Applications in Large-scale Pervasive Systems. In *8th Int'l Conf. on Wireless and Mobile Computing, Networking and Communications*, pages 142–149. IEEE, 2012.