Share service: a design pattern for the dynamic services composition

Rosario Culmone*

[a]*School of Science and Technology, University of Camerino, Camerino, Italy*

Abstract

There are millions of home automation devices in the world and billions in the future. Many of these interact with difficulty due to the absence of a standard. In this paper, we suggest a vision of how we can overcome the problems of interaction in pear-to-pear way. Instead of looking for a network standard, we propose a programming standard through a design pattern. The proposed pattern design does not travel on data network but code. This feature allows not only to exceed the limits of data-oriented standards but to make up for services on the fly. Verification mechanisms that guarantee correctness control the dynamic composition. Finally, an example is presented using the Lua programming language.

Keywords: `Design pattern, Metaprogramming, Dynamic composition, Pear-to-pear`

1. Introduction

I dispositivi IoT [1] connessi nel 2016 erano circa 16 miliardi e saranno 29 miliardi nel 2022 con una crescita annua di circa il 20%. Circa il 20% sono per home automation o short range IoT [2]. I dispositivi di home automation sono installati nel tempo e variano in numero e tipologia senza un preordinato piano di installazione. Mentre i dispositivi short range, come orologi, occhiali, indumenti, variano a seconda delle esigenze degli utenti. Fornire servizi che

*Corresponding author
URL: `rosario.culmone@unicam.it` (Rosario Culmone)

comprendono la loro collaborazione è difficile poiché protocolli di interconnessione e di interazione dei servizi sono diversi e possono variare nel tempo. La standardizzazione in questo settore è difficile poiché la varietà dei servizi è molto ampia. Si va dai sistemi di allarme agli elettrodomestici o ai dispositivi indossabili ed ogni tipologia di dispositivo ha le sue peculiarità. Molto si è fatto in questo settore per cercare di uniformare e permettere l'interazione ad esempio utilizzando ontologie [3]. La situazione per quanto riguarda i layers data link, network, transport e session è ancora indefinita essendo decine gli standard utilizzati [4]. Per il layer 7, application, vi sono diverse proposte [5]. Inoltre grandi aziende [6] stanno investendo molto nella gestione dei dati provenienti da IoT. A questo bisogna considerare che la complessità e specificità dei servizi rende conveniente l'edge computing [7] rispetto al cloud o approcci misti come in [8]. In questo contesto il ruolo della gestione dei servizi diventa rilevante [9] e la dinamicità gioca un ruolo importantissimo per l'affidabilità, stabilità e correttezza dei servizi forniti dai dispositivi IoT [10].

La composizione on the fly dei servizi è un problema affrontato già da diverso tempo [11, 12]. Le proposte esistenti si basano sulla possibilità di invocare servizi a richiesta.Tuttavia esiste un'altro tipo di dinamicità che dipende dal tipo e numero di dispositivi disponibili in un determinato istante. Inoltre può essere utile modificare la semantica di un servizio a seconda dei dispositivi disponibili in un determinato momento. Il punto focale è il modo con cui si determina la compatibilità tra chi chiama il servizio e il servizio fornito. Gli approcci basati sulla definizione di un protocollo orientato ai dati come WSDL [13, 14, 15] che usano validatori [16] per il controllo della compatibilità sono complessi e difficilmente realizzabili su sistemi embedded. Inoltre complessi vincoli sugli schemi non possono essere verificati in WSDL e necessitano di validatori basati sulla logica del primo ordine [17, 18, 19]. Diverso approccio è quello basato su eventi asincroni. Tuttavia i diversi modelli, quelli sincroni basati su RPC [20] e quelli asincroni come pub/sub [21] impongono un approccio metodologico "a priori". La nostra proposta lascia completa libertà sul modello di interazione di basso livello prediligendo la modularità e composizione dei servizi. Esistono proposte

molto ardite su composizione dinamica e interazione di dispositivi [22] ma soffrono di "gigantismo" talvolta ostativo per l'applicabilità in sistemi piccoli.

Quindi in ordine alla corretta verifica dei vincoli definiti dal chiamante e dal chiamato, ci sembra che un approccio basato sulla progettazione by contract sia la più adeguata [23]. La verifica che le specifiche del chiamato siano compatibili con quelle del chiamante può essere verificata usando la Satisfiability Modulo Theories. Vi sono già diversi strumenti come Boogie [24] e Z3 [25] che permettono una efficiente verifica. Tuttavia, poiché la verifica deve essere effettuata molto rapidamente, una soluzione può essere la verifica a posteriori delle specifiche del chiamante. In pratica l'invocazione viene effettuata comunque verso i servizi che esplicitano una affinità semantica. Se il fallimento è nella fase di esecuzione, vuol dire che le precondizioni del chiamante non soddisfano quelle del chiamato. Se il fallimento avviene nella verifica delle post condizioni del chiamante vuol dire che le post condizioni del chiamato non soddisfano le post condizioni del chiamante. In ogni caso devono essere utilizzati strumenti che supportano il controllo del fallimento dell'esecuzione del codice.

OSGi è un framework che ha avuto grande successo nella definizione di uno standard per la produzione di servizi sotto forma di "bundle". Necessità di una piattaforma abbastanza potente per poter eseguire una JVM, i "dundle" e i servizi della piattaforma OSGi. Un dispositivo IoT mediamente "stupido" ha per lo più risorse per essere al più un client MQTT. Quindi l'architettura eterogenea che prevede dispositivi con limitate risorse necessita di diversi protocolli, piattaforme, gateway e framework. Un dispositivo realizzato con un microcontrollore ESP32 [26] che costa appena 2$ e che ha a bordo una connessione wifi non può ospitare un framework OSGi ma solo un client MQTT. Quindi per realizzare servizi OSGi con questo dispositivi bisogna prevedere un server MQTT un gateway che trasformi eventi MQTT in servizi OSGi e quindi realizzare servizi OSGi che realizzino le funzionalità richieste. Inoltre se il dispositivo dispone di attuatori che devono essere controllati allora MQTT non basta. Quindi altri protocolli come SNMP devono essere installati sui dispositivi rentendo ancora più onerosa la gestione. Se il dispositivo è abbastanza potente da ospitare una

JVM e il framework id OSGi (circa 400KB), allora si possono sfruttare appieno le funzionalità di OSGi.

Viceversa un dispositivo IoT basato su ESP32 [27]può direttamente eseguire codice LUA e può implementare il servizio mettendolo a disposizione di altri dispositivi IoT implementandolo con il pattern proposto Share.

Inoltre ci sono due problemi che bisogna considerare se si usa OSGi. Innanzitutto, le API OSGi non dovrebbero utilizzare classi che non sono disponibili su tutti gli ambienti (dispositivi). In secondo luogo, un bundle non dovrebbe avviarsi se contiene codice che non è disponibile nell'ambiente di esecuzione (dispositivo). Questi problemi sono stati risolti ma a costo di onerose operazioni dinamiche che pesano inevitabilmente sulle prestazioni di piccoli dispositivi.

Un'altro importante aspetto riguarda le modalità di interconnesione dei dispositivi. La modalità offerta sino ad ora di sistemi wifi con access point giustificava la presenza di un dispositivo con prestazioni adeguate per fornire servizi onerosi come broker MQTT o framework OSGi. Ma la necessità di espandere l'area coperta da dispositivi wifi o l'eterogeneità dei dispositivi e dei domini applicativi ha spinto a considerare le reti mesh come valida alternativa alle usuali reti con access point [28]. Quindi da qui la necessità di poter utilizzare uno strumento di programmazione pear-to-pear che sfrutti le potenzialità delle reti mesh e che offra servizi basati sulla composizione dinamica.

2. Design pattern

2.1. Class diagram

Il design pattern [29] *Share* è composto da tre classi *Share*, *Service* e *Feature*. *Share* gestisce lo spazio comune per la condivisione dei servizi e *Service* con *Feature* l'implementazione del servizio. Ogni servizio deve sottoscriversi ad almeno un fornitore del servizio di *Sharing* per consentire ad altri di usufruire del servizio. Un servizio possiede una stringa identificativa, ad esempio un codice MIB del protocollo SNMP [30]. Ogni servizio implementa un predicato *pre* che specifica le precondizioni del servizio realizzato mediante *function*. Il pattern

share possiede una componente pubblica, che deve essere implementata con un linguaggio comune a tutti i servizi che condivido o che utilizzano servizi, ed una componente privata che può essere realizzata in un linguaggio differente. In pratica tutte le operazioni e dati che *share* gestisce ad esclusione di *daemon* devono essere realizzate con un comune linguaggi, che come vedremo è sarà scelto tra i linguaggi metaprogrammabili. Il linguaggio con cui è implementato *function* è comune a tutti i servizi che sono condivisi di modo che possa essere spostato da una piattaforma di esecuzione ad un'altra. Tuttavia *function* è una interfaccia molto leggera funge da interfaccia verso il corpo realizzato da *daemon*. L'operazione *daemon* può essere realizzata nello stesso linguaggio di *function* o parzialmente o totalmente in un diverso linguaggio.

Nell'implementazione di *function* possono essere richiesti servizi da altre piattaforme mediante l'invocazione di *call*. Si tratta di uno stub che cerca, verifica, esegue e produce i risultati relativi ad un servizio remoto. I parametri che sono passati a *call* sono un'espressione regolare e i parametri su cui effettuare l'operazione. La funzione *call* produce un booleano che indica se l'operazione è andata a buon fine e una tupla con i valori prodotti dal servizio richiesto.

Ogni servizio realizza la funzione mediante due blocchi di codice *function* e *daemon*. Il codice *function* è il codice che viene inviato al richiedete il servizio e realizza l'interfaccia di comunicazione con il suo interlocutore *daemon* che invece viene eseguito da chi fornisce il servizio. Le due operazioni *function* e *daemon* stabiliscono il tipo, modalità e protocolli della comunicazione le quali non fanno parte del pattern. Ad esempio possono implementare una PRC mediante CORBA [31] o 0MQ [32] Tuttavia poiché l'esecuzione del modulo *function* avviene sul richiedente bisogna che tutti i dispositivi che cooperano possano eseguire il codice di *function*. Quindi il pattern richiede che tutti i componenti che realizzano il pattern usino lo stesso linguaggio per il codice *function*. Naturalmente è possibile diverse implementazioni, mediante design by aspect [33], che tengano conto delle diverse piattaforme di esecuzione. Per quanto riguarda il codice *daemon* questo non essendo soggetto ad essere mosso può essere scritto in qualsiasi linguaggio di programmazione che possa essere eseguito sul dispositivo che lo ospita. Questa caratteristica del codice *daemon* permette di tutelare brevetti su algoritmi e codice che partecipa alla fornitura del servizio. Inoltre usualmente i driver di dispositivi di I/O sono realizzati anche in assembler o con

linguaggi efficienti. Questi driver sono codici *daemon*.

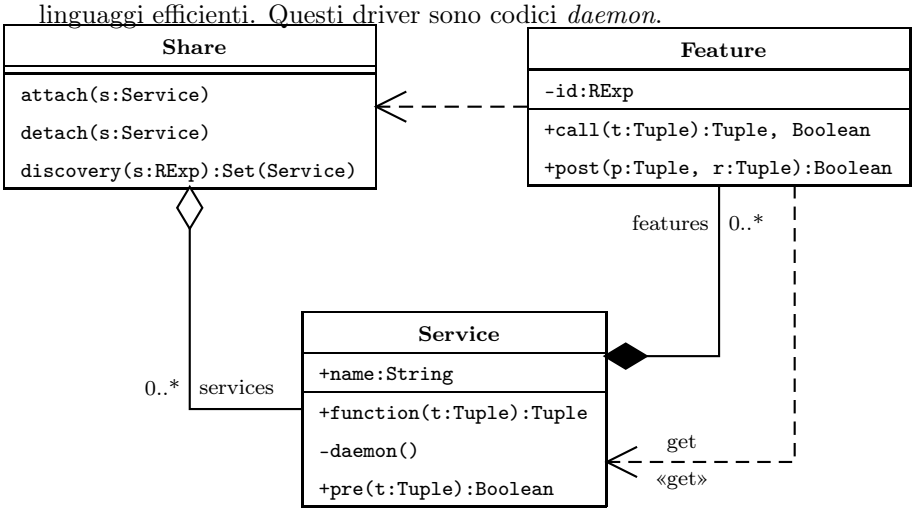

Figure 1: Class diagram

Il pattern *share* è un pattern di metaprogrammazione poiché alcune parti del servizio sono note solo al momento dell'esecuzione e dipendono dallo stato del sistema. La definizione di un servizio richiede la codifica di una funzione *function* e della sua eventuale componente residente *daemon*. Un predicato *pre* specifica le precondizioni per *function*. Un attributo di tipo stringa identifica il servizio in mondo univoco. Ad esempio una stringa MIB SNMP può essere utilizzata per identificare un servizio. Nella codifica di *function* vi possono esservi chiamate a servizi esterni che sono specificati mediante implementazioni di *Feature*. L'attributo *id* definisce un'espressione regolare che descrive semanticamente il servizio richiesto. L'invocazione di un servizio all'interno di *function* avviene mediante l'invocazione di *call* relativa ad una specifica presente in *features*. L'operazione *call* invoca *discovery* con attributo *id* per individuare tutti i servizi sottoscritti a *Share* a cui il servizio chiamante è sottoscritto. Se viene prodotto un insieme non nullo di servizi, viene avviata la fase successiva di invocazione dei servizi. Per questa fase viene utilizzata la primitiva *get* per ottenere codice da un servizio. Questa funzionalità è richiesta esplicitamente

per l'implementazione del pattern. Nel pattern viene indicata una associazione stereotipata *«get»* per indicare il canale di comunicazione con cui verrà inviato codice a richiesta. In particolare viene richiesto il codice *pre* relativa alle precondizioni della funzione relativa al servizio. Una volta trasferita sul chiamante il predicato *pre* viene eseguito applicato ai parametri del chiamante. Se il predicato ha esito positivo, il chiamante richiede mediante *get* il codice *function* altrimenti si ripete l'operazione con il servizio successivo prodotto da *Share*. Una volta ricevuto il codice *function* viene subito invocato sul chiamante mentre il codice *daemon* viene eseguito sul chiamato. La funzione *daemon* è il codice locale al servizio che viene eseguito localmente e può essere realizzato con un linguaggio differente da *function* . L'esecuzione di *function* avviene parallelamente a *daemon* e tra i due processi possono avvenire interazione con protocolli proprietari. La terminazione di *function* sul chiamante produce una tupla con il risultato del calcolo. Viene quindi applicato su tale tupla la postcondizione *post* associata la servizio richiesto. Se il predicato *post* è verificato, la tupla prodotta da *function* è a disposizione del chiamato altrimenti si ripete con il successivo servizio identificato da *Share*. La chiamata di *call* produce *false* se tutte le invocazioni di *Service::pre* producono *false* o tutte le invocazioni di *Feature::post* producono *false*. L'operazione *Feature::call(e:RegEx,t:Sequence):Boolean,Sequence* produce un valore booleano per segnalare il successo o insuccesso dell'invocazione di un servizio con specifica semantica *e* e parametri *t*. E' possibile definire servizi senza parametri o che non producono valori. In questo caso non verranno applicate precondizioni o postcondizioni ma il valore booleano segnalerà anomalie nell'esecuzione dei servizi invocati. Nella specifica di *Feature::post(p:Sequence, t:Sequence) p* sono i parametri del servizio richiesto mentre *t* sono i valori calcolati da *function*. Sono ambedue passati a *post* per permettere di effettuare controlli incrociati tra parametri e valori calcolati. Una dei controlli standard che dovrebbero essere effettuati riguarda il tipo ed il numero dei parametri richiesi e prodotti dalla funzione.

```
contex Feature::call(t:Service)
  pre: service->excludes(s)
```

```
  post: services->includes(s)
```

```
contex Share::attach(s:Service)
  pre: service->excludes(s)
  post: services->includes(s)
```

```
contex Share::detach(s:Service)
  pre: service->includes(s)
  post: services->excludes(s)
```

```
contex Share::discovery(s:String):Set(Service)
  post: result = Set(services->select(name.matches(s)))
```

Listing 1: call function

```
contex Feature::call(t:Tuple):Boolean, Tuple
   def: found : Set(Service) =
     select(s : Share.discovery(id) |
       let s.pre(k:Tuple):Boolean, self.post(v,q):Boolean, s.function(w):r in
          t.isOclType() = k.isOclType() and
          t.isOclType() = v.isOclType() and
          r.isOclType() = q.isOclType() and
          s.pre(t) and self.post(t,s.function(t)))
   post: if found->notEmpty() --insert check closure
     then result = true,found->first().function(t)
     else result = false,Sequence{} endif
```

La struttura del pattern definisce in modo in modo rigoroso il comportamento di tre classi e delle loro operazioni. Per quanto riguarda *Share* la semantica delle operazioni è definita completamente. Per la classe *Service* è definita la firma delle operazioni *function, pre e daemon.* Per la classe *Feature* è definita la firma di *post.* Per l'operazione *call* è definito in modo rigoroso il comportamento [**?**] ma non la funzionalità essendo il cuore del meccanismo di metraprogrammazione del pattern. L'implementazione richiede la definizione dell'operatore *get* che permette la richiesta e l'invio di codice. Il pattern non specifica come questo debba essere fatto ma richiede la sua realizzazione. E' evidente che l'implementazione del pattern da parte dei nodi di elaborazione richiede che tutti condividano il protocollo di rete che realizza la primitiva *get.*

L'uso del pattern può essere realizzato definendo una classe concreta *Share* e due classi astratte *Service* e *Feature*. Ad esempio in Java *function*, *pre*, *post* possono essere realizzate come lambda expression. I linguaggi come perl, pyton, ruby, lua, php, object C che ammettono metaprogrammazione l'implementazione e molto semplice. Nell'implementazione bisogna fare attenzione alla gestione dei parametri passati a *pre, post e function*. Anche se nella specifica di *call* 1 vi è corrispondenza tra numero e tipo dei parametri è buona norma che in *pre* si verifichino numero e tipo di parametri passati così come in *post*.

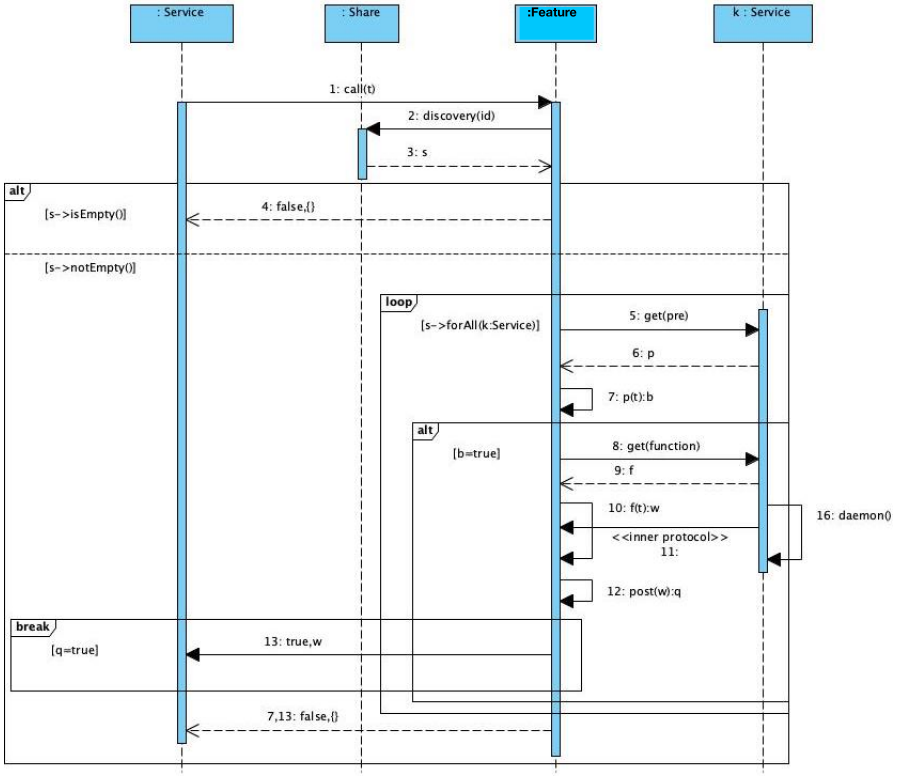

Figure 2: The call service

Quindi mentre la classe *Share* può essere realizzata direttamente, il pattern *Share* specifica il comportamento dla realizzazione del pattern richiede sostanzialmente l'implementazione di *function, daemon, pre e post* utilizzando immutate le funzionalità delle rimanenti operazioni legate alla struttura del

pattern. Nella struttura di pattern, la sottoscrizione del servizio a *Share* non pone nessun vincolo. Quindi è possibile sottoscriversi a più *Share* o a solo uno. Un'altro aspetto è l'interazione tra *daemon* e *function*. Il codice associato a *function* viene ricevuto mediante l'operatore *get* subito dopo eseguito con *daemon*. L'interazione tra *function* e *daemon* avviene mediante protocolli non previsti dal pattern. Ad esempio può essere utilizzato il protocollo come *0MQ* [**?**]. La corretta interazione tra *function* e *daemon* è garantita dal fatto che ambedue sono sviluppati dallo stesso programmatore.

L'operazione di *attach* serve per registrarsi ad un sottoscrittore. Ogni dispositivo può essere sottoscrittore tuttavia è plausibile che solo pochi forniscano questo servizio.

La ricerca di un servizio avviene mediante una stringa descrittiva. Potrebbe essere un descrittore ontologico o semplicemente una stringa MIB del protocollo SNMP. Nella specifica si è utilizzata la funzione *match*

Il pattern permette la chiamata annidata di servizi. Questo comporta un problema se si forma un ciclo di invocazione essendo la presenza del ciclo verificabile solo a tempo di esecuzione (non esiste ciclo strutturato). La soluzione consiste nello sfruttare il nome unico del servizio *name* per creare una lista di invocazione passata da chi invoca il servizio al servizio invocata. La specifica di *call* diventa

```
contex Feature::call(t:Tuple,s:Sequence):Tuple
--and s->excludes(found->first().name)
```

e l'invocazione diventa `call(t:Tuple,s->append(self.name))`

Infine l'esecuzione del codice *resident* del servizio effettuerà, a secondo del flusso di esecuzione, le invocazioni del codice *nomad* che a loro volta innescheranno le interazioni con il codice *resident* del servizio chiamato (figura **??**).

3. Example

Consideriamo un ambiente domestico in cui sono presenti gli usuali dispostivi per il riscaldamento ovvero caldaia, termostati e termometri. Se ogni dispos-

itivo implementa il pattern *Share* si potranno avere comportamenti diversi a seconda della numerosità dei vari dispositivi. Ad esempio i termometri possono essere su una stanza o su più stanze o su tutte le stanze come i termostati. Può esservi solo una caldaia o più caldaie. Inoltre l'impianto può essere in una casa unifamiliare, bifamiliare o in appartamenti. Ogniuna di queste possibili configurazioni può produrre comportamenti diversi sulle modalità e tipo di riscaldamento. Ad esempio su una casa unifamiliare composta da una caldaia, un termometro e un termostato la logica è di accendere la caldaia se la temperatura della stanza scende al di sotto del valore impostato sul termostato. Ma se il numero di termometri diventano più di uno la caldaia verrà accesa se la media delle temperature misurate dai termometri e al di sotto della temperatura impostata dal termostato. Se inoltre i termostati diventano più di uno allora sarà la caldaia a decidere se accendere quando almeno uno dei termostati avrà dato l'abilitazione. Tutte queste logiche possono essere sviluppate sui singoli dispositivi che hanno competenza sulle azioni che intraprendono secondo le diverse configurazioni che vengono rilevate permettendo flessibilità, adattamento e fault tolerance la sistema.

4. Performance

Complessità analitica e simulatore e risultati simulatore

5. Future works

Realizzazione di un linguaggio di dominio per sistemi IoT. Analisi statica delle performance del sistema. Integrazione con cloud di dati.

6. Bibliography styles

7. Appendice

```
pcall()
```

References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer Networks 54 (15) (2010) 2787 – 2805. doi:10.1016/j.comnet.2010.05.010.

[2] E. M. Report, Internet of things forecast (2019).
URL https://www.ericsson.com/en/mobility-report/internet-of-things-forecast

[3] M. Bermudez-Edo, T. Elsaleh, P. Barnaghi, K. Taylor, Iot-lite ontology (2015).
URL http://www.w3.org/Submission/2015/SUBM-iot-lite-20151126

[4] R. Jain, Internet of things protocols and standards (2015).
URL https://www.cse.wustl.edu/~jain/cse570-15/ftp/iot_prot

[5] A. P. Castellani, M. Gheda, N. Bui, M. Rossi, M. Zorzi, Web services for the internet of things through coap and exi, in: 2011 IEEE International Conference on Communications Workshops (ICC), 2011, pp. 1–6. doi:10.1109/iccw.2011.5963563.

[6] T. Pflanzner, A. Kertesz, A survey of iot cloud providers, in: 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016, pp. 730–735. doi:10.1109/MIPRO.2016.7522237.

[7] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu, Edge computing: Vision and challenges, IEEE Internet of Things Journal 3 (5) (2016) 637–646. doi:10.1109/JIOT.2016.2579198.

[8] M. Villari, M. Fazio, S. Dustdar, O. Rana, R. Ranjan, Osmotic computing: A new paradigm for edge/cloud integration, IEEE Cloud Computing 3 (6) (2016) 76–83. doi:10.1109/MCC.2016.124.

[9] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings

of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.
URL http://doi.acm.org/10.1145/2342509.2342513

[10] B. Cheng, M. Wang, S. Zhao, Z. Zhai, D. Zhu, J. Chen, Situation-aware dynamic service coordination in an iot environment, IEEE/ACM Transactions on Networking 25 (4) (2017) 2082–2095. doi:10.1109/TNET.2017.2705239.

[11] H. Pourreza, P. Graham, On the fly service composition for local interaction environments, in: Fourth Annual IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOMW'06), 2006, pp. 6 pp.–399. doi:10.1109/PERCOMW.2006.104.

[12] Q. Zhao, G. Huang, J. Huang, X. Liu, H. Mei, A web-based mashup environment for on-the-fly service composition, in: 2008 IEEE International Symposium on Service-Oriented System Engineering, 2008, pp. 32–37. doi:10.1109/SOSE.2008.9.

[13] D. Booth, C. K. Liu, Web services description language (wsdl) version 2.0 part 0: Primer (2007).
URL http://www.w3.org/TR/wsdl20-primer

[14] R. Chinnici, J.-J. Moreau, A. Ryman, S. Weerawarana, Web services description language (wsdl) version 2.0 part 1: Core language (2007).
URL http://www.w3.org/TR/wsdl20

[15] R. Chinnici, H. Haas, A. A. Lewis, J.-J. Moreau, D. Orchard, S. Weerawarana, Web services description language (wsdl) version 2.0 part 2: Adjuncts (2007).
URL http://www.w3.org/TR/wsdl20-adjuncts

[16] E. Marchetti, C. Bartolini, A. Bertolino, A. Polini, Ws-taxi: A wsdl-based testing tool for web services, in: 2009 International Conference on Software Testing Verification and Validation(ICST), Vol. 00, 2009, pp. 326–335. doi:10.1109/ICST.2009.28.
URL doi.ieeecomputersociety.org/10.1109/ICST.2009.28

[17] D. Cacciagrano, F. Corradini, R. Culmone, L. Vito, Dynamic constraint-based invocation of web services, in: Web Services and Formal Methods, Third International Workshop, WS-FM 2006 Vienna, Austria, September 8-9, 2006, Proceedings, 2006, pp. 138–147. doi:10.1007/11841197_9.

[18] D. Cacciagrano, F. Corradini, R. Culmone, L. Tesei, L. Vito, A model-prover for constrained dynamic conversations, in: iiWAS'2008 - The Tenth International Conference on Information Integration and Web-based Applications Services, 24-26 November 2008, Linz, Austria, 2008, pp. 630–633. doi:10.1145/1497308.1497428.

[19] D. Cacciagrano, F. Corradini, R. Culmone, L. Vito, Constraint-based dynamic conversations, in: The Fifth International Conference on Networking and Services, ICNS 2009, 20-25 April 2009, Valencia, Spain, 2009, pp. 7–12. doi:10.1109/ICNS.2009.55.

[20] J. Bloomer, Power Programming with RPC, O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[21] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Comput. Surv. 35 (2) (2003) 114–131. doi:10.1145/857076.857078.
URL http://doi.acm.org/10.1145/857076.857078

[22] R. Baldoni, C. Ciccio, M. Mecella, F. Patrizi, L. Querzoni, G. Santucci, S. Dustdar, F. Li, H.-L. Truong, L. Albornos, F. Milagro, P. Antolin Rafael, R. Ayani, K. Rasch, M. Garcia Lozano, M. Aiello, A. Lazovik, A. Denaro, G. Lasala, P. Pucci, C. Holzner, F. Cincotti, F. Aloise, An Embedded Middleware Platform for Pervasive

and Immersive Environments for-All, University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science, 2009, relation: https://www.rug.nl/informatica/onderzoek/bernoulli Rights: University of Groningen, Johann Bernoulli Institute for Mathematics and Computer Science.

[23] B. Meyer, Object-oriented Software Construction (2Nd Ed.), Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[24] M. Barnett, R. Leino, Weakest-precondition of unstructured programs, in: PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, ACM Press, New York, NY, USA, 2005, pp. 82–87.

[25] L. de Moura, N. Bjørner, Z3: An efficient smt solver, in: C. R. Ramakrishnan, J. Rehof (Eds.), Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 337–340.

[26] Espressif, Esp32 soc (2019).
URL `https://www.espressif.com/`

[27] whitecat, Whitecat esp32 n1 board (2019).
URL `https://whitecatboard.org/lorawan-deployment-in-cornella/`

[28] L. Li, H. Xiaoguang, C. Ke, H. Ketai, The applications of wifi-based wireless sensor network in internet of things and smart grid, in: Industrial Electronics and Applications (ICIEA), 2011 6th IEEE Conference on, IEEE, 2011, pp. 789–793.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, USA, 1994.

[30] L. Andrey, O. Festor, A. Lahmadi, A. Pras, J. Schönwälder, Survey of snmp performance analysis studies, International Journal of Network Management 19 (6) (2009) 527–548, 10.1002/nem.729. `doi:10.1002/nem.729`.

[31] O. M. Group, Orba component model (2006).
URL https://www.omg.org/spec/CCM

[32] P. Hintjens, Zeromq message transport protocol (2019).
URL https://rfc.zeromq.org/spec:23/ZMTP

[33] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings, 1997, pp. 220–242. doi:10.1007/BFb0053381.
URL https://doi.org/10.1007/BFb0053381