FioranoMQ® 9

Concept Guide

FIORANO END-USER LICENSE AGREEMENT

THIS FIORANO END-USER LICENSE AGREEMENT (THE AGREEMENT) IS A LEGAL AGREEMENT BETWEEN YOU (HEREINAFTER CUSTOMER), EITHER AN INDIVIDUAL OR A CORPORATE ENTITY, AND FIORANO SOFTWARE, INC., HAVING A PLACE OF BUSINESS AT 718 UNIVERSITY AVE, SUITE 212 LOS GATOS, CA 95032, USA, OR ITS AFFILIATED COMPANIES (HEREINAFTER FIORANO) FOR CERTAIN SOFTWARE DEVELOPED AND MARKETED BY FIORANO AS DEFINED IN GREATER DETAIL BELOW. BY OPENING THIS PACKAGE, INSTALLING, COPYING, DOWNLOADING, EXTRACTING AND/OR OTHERWISE USING THE SOFTWARE, YOU ARE CONSENTING TO BE BOUND BY AND ARE BECOMING PARTY TO THIS AGREEMENT ON THE DATE OF INSTALLATION, COPYING, DOWNLOAD OR EXTRACTION OF THE SOFTWARE (THE EFFECTIVE DATE). IF YOU DO NOT AGREE WITH ANY OF THE TERMS OF THIS AGREEMENT, PLEASE STOP INSTALLING AND/OR USING THE SOFTWARE AND PROMPTLY RETURN THE UNUSED SOFTWARE TO THE PLACE OF PURCHASE. BY DEFAULT, THE SOFTWARE IS MADE AVAILABLE TO CUSTOMERS IN ONLINE, DOWNLOADABLE FORM. THE TERMS OF THIS AGREEMENT SHALL APPLY TO EACH SOFTWARE LICENSE GRANTED BY FIORANO UNDER THIS AGREEMENT.

1. DEFINITIONS

a. **Affiliate** means, in relation to Fiorano, another person firm or company which directly or indirectly controls, is controlled by or is under common control with Fiorano and the expression 'control' shall mean the power to direct or cause the direction of the general management and policies of the person firm or company in question.

b. **Commencement Date** means the date on which Fiorano delivers the Software to Customer, or if no delivery is necessary, the Effective Date set forth in this Agreement or on the relevant Order Form.

c. **Designated Center** means the computer hardware, operating system, customer-specific application and Customer Geographic Location at which the Software is deployed as designated on the corresponding Order Form.

d. **Designated Contact** shall mean the contact person or group designated by Customer and agreed to by Fiorano who will coordinate all Support requests to Fiorano.

e. **Documentation** means the user guides and manuals for installation and use of the Software. Documentation is provided in CD-ROM or bound form, whichever is generally available.

f. **Error** shall mean a reproducible defect in the Supported Program or Documentation when operated on a Supported Environment which causes the Supported Program not to operate substantially in accordance with the Documentation.

g. **Excluded Components** shall mean such components as are listed in Exhibit B. Such Excluded Components do not constitute Software under this Agreement and are third party components supplied subject to the corresponding license agreements specified in Exhibit B.

h. **Excluded License** shall mean and include any license that requires any portion of any materials or software supplied under such license to be disclosed or made available to any party either in source code or object code form. In particular, all versions and derivatives of the GNU GPL and LGPL shall be considered Excluded Licenses for the purposes of this Agreement.

i. **Resolution** shall mean a modification or workaround to the Supported Program and/or Documentation and/or other information provided by Fiorano to Customer intended to resolve an Error.

j. **Residuals** shall mean information in non-tangible form which may be retained by persons who have had access to the Confidential Information, including ideas, concepts, know-how or techniques contained therein.

k. **Order Form** means the document in hard copy form by which Customer orders Software licenses and services, and which is agreed to in writing by the parties. The Order Form shall reference the Effective Date and be governed by the terms of this Agreement. Customer understands that any document in the nature of a purchase order originating from Customer shall not constitute a contractual offer and that the terms thereof shall not govern any contract to be entered into between Fiorano and Customer. The Order Form herein shall constitute an offer to purchase made by the Customer under the terms of the said Order Form and this Agreement.

l. **Software** means each of the individual Products, as further outlined in Exhibit-A, in object code form distributed by Fiorano for which Customer is granted a license pursuant to this Agreement, and the media, Documentation and any Updates thereto.

m. **Support** shall mean ongoing support provided by Fiorano pursuant to the terms of this Agreement and Fiorano's current support policies. **Supported Program or Supported Software** shall mean the then current version of the Software in use at the Designated Center for which the Customer has paid the then-current support fee **(Support Fee)**.

n. **Support Hours** shall mean 9 AM to 5 PM, Pacific Standard Time, Monday through Friday, for Standard Support.

o. **Support Period** shall mean the period during which Customer is entitled to receive Support on a particular Supported Program, which shall be a period of twelve (12) months beginning from the Commencement Date, or if applicable, twelve (12) months from the expiration of the preceding Support Period. Should Fiorano withdraw support pursuant to section 1 (q), the Support Period shall be automatically reduced to the expiration date of the appropriate Software.

p. **Supported Environment** shall mean any hardware and operating system platform which Fiorano provides Support for use with the Supported Program.

q. **Update** means a subsequent release of the Software that Fiorano generally makes available for Supported Software licensees at no additional license fee other than shipping and handling charges. Update shall not include any release, option, feature or future product that Fiorano licenses separately. Fiorano will provide Updates for the Supported Programs as and when developed for general release in Fiorano's sole discretion. Fiorano may withdraw support for any particular version of the Software, including without limitation the most current Update and any preceding release with a notice of three (3) months to Customer.

2. SOFTWARE LICENSE.

(a) Rights Granted, subject to the receipt by Fiorano of appropriate license fees.

(i) The Software is Licensed to Customer for use under the terms of this Agreement and **NOT SOLD**. Fiorano grants to Customer a limited, non-exclusive, world wide license to use the Software as specified on an Order Form and subject to the licensing restrictions in Exhibit C under this Agreement, as follows:

(1) to use the Software solely for Customer's operations at the Designated Center consistent with the use limitations specified or referenced in this Agreement, the Documentation for such Software or any Order Form accepted by Fiorano pursuant to this Agreement. Customer may not relicense, rent or lease the Software or use the Software for third party training, commercial timesharing or service bureau use;

(2) to use the Documentation provided with the Software in support of Customer's authorized use of the Software;

(3) to make a single copy for back-up or archival purposes and/or temporarily transfer the Software in the event of a computer malfunction. All titles, trademarks and copyright or other restricted rights notices shall be reproduced in any such copies;

(4) to allow third parties to use the Software for Customer's operations, so long as Customer ensures that use of the Software is in accordance with the terms of this Agreement.

(ii) Customer shall not copy or use the Software (including the Documentation) except as specified in this Agreement and applicable Order Form. Customer shall have no right to use other third party software or Excluded Components that are included within the Software except in connection and within the scope of Customer's use of Fiorano's Software product.

(iii) Customer agrees not to cause or permit the reverse engineering, disassembly, decompilation, or any other attempt to derive source code from the Software, except to the extent expressly provided for by applicable law.

(iv) Customer hereby warrants that it shall not, by any act or omission, cause or permit the Products or any part thereof to become expressly or impliedly subject to any Excluded License.

(v) Fiorano and its Affiliates shall retain all title, copyright and other proprietary rights in the Software. Customer does not acquire any rights, express or implied, in the Software, other than those specified in this Agreement.

(vi) Customer agrees that it will not publish or cause or permit to be published any results of benchmark tests run on the Software.

(vii) If the Software is licensed for a specific term, as noted on the Order Form, then the license shall expire at the end of the term and the termination conditions in section 4(d) shall automatically become applicable.

(b) Transfer. Customer may transfer a Software license within its organization upon notice to Fiorano; transfers are subject to the terms and fees specified in Fiorano's transfer policy in effect at the time of the transfer. If the Software is licensed for a specific term, then it may not be transferred by Customer.

(c) Verification. At Fiorano's written request, Customer shall furnish Fiorano with a signed certification verifying that the Software is being used pursuant to the provisions of this Agreement and applicable /Order Form. Fiorano (or Fiorano's designee) may audit Customer's use of the Software. Any such audit shall be conducted during regular business hours at Customer's facilities and shall not unreasonably interfere with Customer's business activities. If an audit reveals that Customer has underpaid fees to Fiorano, Customer shall be invoiced directly for such underpaid fees based on the Fiorano Price List in effect at the time the audit is completed. If the underpaid fees are in excess of five percent (5%) of the aggregate license fees paid to Fiorano pursuant to this Agreement, the Customer shall pay Fiorano's reasonable costs of conducting the audit. Audits shall be conducted no more than once annually.

(d) Customer Specific Objects.

(i) The parties agree and acknowledge, subject to Fiorano's underlying proprietary rights, that Customer may create certain software objects applicable to Customer's internal business (Customer Specific Objects). Any Customer Specific Object developed solely by Customer shall be the property of Customer. To the extent that Customer desires to have Fiorano incorporate such Customer Specific Objects into Fiorano's Software (and Fiorano agrees, in its sole discretion, to incorporate such Customer Specific Objects), Customer will promptly deliver to Fiorano the source and object code versions (including documentation) of such Customer Specific Objects, and any updates or modifications thereto, and hereby grants Fiorano a perpetual, irrevocable, worldwide, fully-paid, royalty-free, exclusive, transferable license to reproduce, modify, use, perform, display, distribute and sublicense, directly and indirectly, through one or more tiers of sublicensees, such Customer Specific Objects.

(ii) Any objects, including without limitation Customer Specific Objects, developed solely or jointly with Customer by Fiorano shall be the property of Fiorano.

(e) Additional Restrictions on Use of Source Code.

Customer acknowledges that the Software, its structure, organization and any human-readable versions of a software program (Source Code) constitute valuable trade secrets that belong to Fiorano and/or its suppliers Source Code Software, if and when supplied to Customer shall constitute Software licensed under the terms of this Agreement and the Order Form. Customer agrees not to translate the Software into another computer language, in whole or in part.

(i) Customer agrees that it will not disclose all or any portion of the Software's Source Code to any third parties, with the exception of authorized employees (Authorized Employees) and authorized contractors (Authorized Contractors) of Customer who (i) require access thereto for a purpose authorized by this Agreement, and (ii) have signed an employee or contractor agreement in which such employee or contractor agrees to protect third party confidential information. Customer agrees that any breach by any Authorized Employees or Authorized Contractors of their obligations under such confidentiality agreements shall also constitute a breach by Customer hereunder.

(ii) Customer shall ensure that the same degree of care is used to prevent the unauthorized use, dissemination, or publication of the Software's Source Code as Customer uses to protect its own confidential information of a like nature, but in no event shall the safeguards for protecting such Source Code be less than a reasonably prudent business would exercise under similar circumstances. Customer shall take prompt and appropriate action to prevent unauthorized use or disclosure of such Source Code, including, without limitation, storing such Source Code only on secure central processing units or networks and requiring passwords and other reasonable physical controls on access to such Source Code.

(iii) Customer shall instruct Authorized Employees and Authorized Contractors not to copy the Software's Source Code on their own, and not to disclose such Source Code to anyone not authorized to receive it.

(iv) Customer shall handle, use and store the Software's Source Code solely at the Customer Designated Center.

(f) Acceptance tested Software

Customer acknowledges that it has, prior to the date of this Agreement, carried out adequate acceptance tests in respect of the Software. Customer's acceptance of delivery of the Software under this Agreement shall be conclusive evidence that Customer has examined the Software and found it to be complete, and in accordance with the Documentation, in good order and condition and fit for the purpose for which it is required.

3. TECHNICAL SERVICES.

(a) Maintenance and Support Services. Maintenance and Support services will be provided under the terms of this Agreement and Fiorano's support policies in effect on the date Support is ordered by Customer. Support services shall be provided from Fiorano's principal place of business or at the Designated Center, as determined in Fiorano's sole discretion. If Fiorano sends personnel to the Designated Center to resolve any Error in the Supported Program, Customer shall pay Fiorano's reasonable travel, meals and lodging expenses.

(b) Consulting and Training Services. Fiorano will, upon Customer's request, provide consulting and training services agreed to by the parties pursuant to the terms of a separate written agreement.

(c) Incidental Expenses. For any on-site services requested by Customer, Customer shall reimburse Fiorano for actual, reasonable travel and out-of-pocket expenses incurred (separate from then current Support Fees).

(d) Reinstatement. Once Support has been terminated by Customer or Fiorano for a particular Supported Program, it can be reinstated only by agreement of the parties.

(e) Supervision and Management. Customer is responsible for undertaking the proper supervision, implementation and management of its use of the Supported Programs, including, but not limited to: (i) assuring proper Supported Environment configuration, Supported Programs installation and operating methods; and (ii) following industry standard procedures for the security of data, accuracy of input and output, and back-up plans, including restart and recovery in the event of hardware or software error or malfunction. Fiorano does not warrant (i) the performance of, or combination of, Software with any third party software, (ii) any implementation of the Software that does not follow Fiorano's delivery methodology, or (iii) any components not supplied by Fiorano.

(f) Training. Customer is responsible for proper training of all appropriate personnel in the operation and use of the Supported Programs and associated equipment.

(g) Access to Personnel and Equipment. Customer shall provide Fiorano with access to Customer's personnel and its equipment during Support Hours. This access must include the ability to dial-in from Fiorano facilities to the equipment on which the Supported Programs are operating and to obtain the same access to the equipment as those of Customer's employees having the highest privilege or clearance level. Fiorano will inform Customer of the specifications of the modem equipment and associated software needed, and Customer will be responsible for the costs and use of said equipment.

(h) Support Term. Upon expiration of an existing Support Period for a particular Supported Program, a new Support Period shall automatically begin for a consecutive twelve (12) month term (Renewal Period) so long as (i) Customer pays the Support Fee within thirty (30) days of invoice by Fiorano; and (ii) Fiorano is still offering Support on such Supported Program.

(i) Annual Support Fees. Annual Support Fees shall be at the rates set forth in the applicable Order Form.

4. TERM AND TERMINATION.

(a) Term. This Agreement and each Software license granted under this Agreement shall continue perpetually unless terminated under this ***Section 4*** (Term and Termination).

(b) Termination by Customer. If the Software is licensed for a specific term as noted on an Order Form, Customer may terminate any Software license at the end of the term; however, any such termination shall not relieve Customer's obligations specified in ***Section 4(d)*** (Effect of Termination).

(c) Termination by Fiorano. Fiorano may terminate this Agreement or any license upon written notice if Customer breaches this Agreement and fails to correct the breach within thirty (30) days of notice from Fiorano.

(d) Effect of Termination. Termination of this Agreement or any license shall not limit Fiorano from pursuing other remedies available to it, including injunctive relief, nor shall such termination relieve Customer's obligation to pay all fees that have accrued or are otherwise owed by Customer under any Order Form. Such rights and obligations of the parties' which, by their nature, are intended to survive the termination of this agreement shall survive such termination. Without limitation to the foregoing, these shall include rights and liabilities arising under Sections ***2 (a)(iii), 2(a)(iv)*** (Rights Granted)***, 2(d)*** (Customer Specific Objects), ***4*** (Term and Termination), ***5*** (Indemnity, Warranties, Remedies), ***6*** (Limitation of Liability), ***7*** (Payment Provisions), ***8*** (Confidentiality) and ***9*** (Miscellaneous) Upon termination, Customer shall cease using, and shall return or at Fiorano's request destroy, all copies of the Software and Documentation and upon Fiorano's request certify the same to Fiorano in writing within thirty (30) days of termination. In case of termination of this Agreement or any license for any reason by either party, Fiorano shall have no obligation to refund any amounts paid to Fiorano by Customer under this Agreement. Further, if Customer terminates the agreement before the expiry of a term for a term-license, then Customer shall be obliged to pay the entire license fee for the entire licensed term.

5. INDEMNITY, WARRANTIES, REMEDIES.

(a) Infringement Indemnity. Fiorano agrees to indemnify Customer against a third party claim that any Product infringes a U.S. copyright or patent and pay any damages finally awarded, provided that: (i) Customer notifies Fiorano in writing within ten (10) days of the claim; (ii) Fiorano has sole control of the defense and all related settlement negotiations; and (iii) Customer provides Fiorano with the assistance, information and authority at no cost to Fiorano, necessary to perform Fiorano's obligations under this ***Section 5*** (Indemnities, Warranties, Remedies). Fiorano shall have no liability for any third party claims of infringement based upon (i) use of a version of a Product other than the most current version made available to the Customer, (ii) the use, operation or combination of any Product with programs, data, equipment or documentation if such infringement would have been avoided but for such use, operation or combination; or (iii) any third party software, except as the same may be integrated, incorporated or bundled by Fiorano, or its third party licensors, in the Product licensed to Customer hereunder.

If any Product is held or claimed to infringe, Fiorano shall have the option, at its expense, to (i) modify the Product to be non-infringing or (ii) obtain for Customer a license to continue using the Software. If it is not commercially reasonable to perform either of the above options, then Fiorano may terminate the license for the infringing Product and refund the pro rated amount of license fees paid for the applicable Product using a twelve (12) month straight-line amortization schedule starting on the Commencement Date. This ***Section 5(a)*** (Infringement Indemnity) states Fiorano's entire liability and Customer's sole and exclusive remedy for infringement.

(B) **WARRANTIES AND DISCLAIMERS.**

(I) SOFTWARE WARRANTY. EXCEPT FOR EXCLUDED COMPONENTS WHICH ARE PROVIDED AS IS WITHOUT WARRANTY OF ANY KIND, FOR EACH SUPPORTED SOFTWARE LICENSE WHICH CUSTOMER ACQUIRES HEREUNDER, FIORANO WARRANTS THAT FOR A PERIOD OF THIRTY (30) DAYS FROM THE COMMENCEMENT DATE THE SOFTWARE, AS DELIVERED BY FIORANO TO CUSTOMER, WILL SUBSTANTIALLY PERFORM THE FUNCTIONS DESCRIBED IN THE ASSOCIATED DOCUMENTATION IN ALL MATERIAL RESPECTS WHEN OPERATED ON A SYSTEM WHICH MEETS THE REQUIREMENTS SPECIFIED BY FIORANO IN THE DOCUMENTATION. PROVIDED THAT CUSTOMER GIVES FIORANO WRITTEN NOTICE OF A BREACH OF THE FOREGOING WARRANTY DURING THE WARRANTY PERIOD, FIORANO SHALL, AS CUSTOMER'S SOLE AND EXCLUSIVE REMEDY AND FIORANO'S SOLE LIABILITY, USE ITS REASONABLE EFFORTS, DURING THE WARRANTY PERIOD ONLY, TO CORRECT ANY REPRODUCIBLE ERRORS THAT CAUSE THE BREACH OF THE WARRANTY IN ACCORDANCE WITH ITS TECHNICAL SUPPORT POLICIES. IF CUSTOMER DOES NOT OBTAIN A SUPPORTED SOFTWARE LICENSE, THE SOFTWARE IS PROVIDED AS IS. ANY IMPLIED WARRANTY OR CONDITION APPLICABLE TO THE SOFTWARE, DOCUMENTATION OR ANY PART THEREOF BY OPERATION OF ANY LAW OR REGULATION SHALL OPERATE ONLY FOR DEFECTS DISCOVERED DURING THE ABOVE WARRANTY PERIOD OF THIRTY (30) DAYS UNLESS TEMPORAL LIMITATION ON SUCH WARRANTY OR CONDITION IS EXPRESSLY PROHIBITED BY APPLICABLE LAW. ANY SUPPLEMENTS OR UPDATES TO THE SOFTWARE, INCLUDING WITHOUT LIMITATION, BUG FIXES OR ERROR CORRECTIONS SUPPLIED AFTER THE EXPIRATION OF THE THIRTY-DAY LIMITED WARRANTY PERIOD SHALL NOT BE COVERED BY ANY WARRANTY OR CONDITION, EXPRESS, IMPLIED OR STATUTORY.

(II) MEDIA WARRANTY. FIORANO WARRANTS THE TAPES, DISKETTES OR ANY OTHER MEDIA ON WHICH THE SOFTWARE IS SUPPLIED TO BE FREE OF DEFECTS IN MATERIALS AND WORKMANSHIP UNDER NORMAL USE FOR THIRTY (30) DAYS FROM THE COMMENCEMENT DATE. CUSTOMER'S SOLE AND EXCLUSIVE REMEDY AND FIORANO'S SOLE LIABILITY FOR BREACH OF THE MEDIA WARRANTY SHALL BE FOR FIORANO TO REPLACE DEFECTIVE MEDIA RETURNED WITHIN THIRTY (30) DAYS OF THE COMMENCEMENT DATE.

(III) SERVICES WARRANTY. FIORANO WARRANTS ANY SERVICES PROVIDED HEREUNDER SHALL BE PERFORMED IN A PROFESSIONAL AND WORKMANLIKE MANNER IN ACCORDANCE WITH GENERALLY ACCEPTED INDUSTRY PRACTICES. THIS WARRANTY SHALL BE VALID FOR A PERIOD OF THIRTY (30) DAYS FROM PERFORMANCE. FIORANO'S SOLE AND EXCLUSIVE LIABILITY AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY PURSUANT TO THIS WARRANTY SHALL BE USE BY FIORANO OF REASONABLE EFFORTS FOR RE-PERFORMANCE OF ANY SERVICES NOT IN COMPLIANCE WITH THIS WARRANTY WHICH ARE BROUGHT TO FIORANO'S ATTENTION BY WRITTEN NOTICE WITHIN FIFTEEN (15) DAYS AFTER THEY ARE PERFORMED.

(IV) DISCLAIMER OF WARRANTIES. **SUBJECT TO LIMITED WARRANTIES PROVIDED FOR HEREINABOVE, AND TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, THE SOFTWARE, DOCUMENTATION AND SERVICES (IF ANY) ARE PROVIDED *AS IS AND WITH ALL FAULTS*, FIORANO HEREBY DISCLAIMS ALL OTHER WARRANTIES AND CONDITIONS, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING, BUT NOT LIMITED TO, ANY (IF ANY) IMPLIED WARRANTIES, DUTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF RELIABILITY OR AVAILABILITY, OF ACCURACY OR COMPLETENESS OF RESPONSES, OF RESULTS, OF WORKMANLIKE EFFORT, OF LACK OF VIRUSES, AND OF LACK OF NEGLIGENCE, ALL WITH REGARD TO THE SOFTWARE, AND THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE OR OTHERWISE ARISING OUT OF THE USE OF THE SOFTWARE. ALSO, THERE IS NO**

WARRANTY OR CONDITION OF TITLE, QUIET ENJOYMENT, QUIET POSSESSION, CORRESPONDENCE TO DESCRIPTION OR NON-INFRINGEMENT WITH REGARD TO THE SOFTWARE.

6. LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL FIORANO BE LIABLE FOR ANY SPECIAL, INCIDENTAL, PUNITIVE, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, BUT NOT LIMITED TO, DAMAGES FOR LOSS OF PROFITS OR CONFIDENTIAL OR OTHER INFORMATION, FOR BUSINESS INTERRUPTION, FOR PERSONAL INJURY, FOR LOSS OF PRIVACY, FOR FAILURE TO MEET ANY DUTY OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE, AND FOR ANY OTHER PECUNIARY OR OTHER LOSS WHATSOEVER) ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE, THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT OR OTHER SERVICES, INFORMATION, SOFTWARE, AND RELATED CONTENT THROUGH THE SOFTWARE, OR OTHERWISE UNDER OR IN CONNECTION WITH ANY PROVISION OF THIS EULA, EVEN IN THE EVENT OF THE FAULT, TORT (INCLUDING NEGLIGENCE), MISREPRESENTATION, STRICT LIABILITY, BREACH OF CONTRACT OR BREACH OF WARRANTY OF FIORANO, AND EVEN IF FIORANO OR ANY SUPPLIER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

NOTWITHSTANDING ANY DAMAGES THAT MAY BE INCURRED FOR ANY REASON AND UNDER ANY CIRCUMSTANCES (INCLUDING, WITHOUT LIMITATION, ALL DAMAGES AND LIABILITIES REFERENCED HEREIN AND ALL DIRECT OR GENERAL DAMAGES IN LAW, CONTRACT OR ANYTHING ELSE), THE ENTIRE LIABILITY OF FIORANO UNDER ANY PROVISION OF THIS EULA AND THE EXCLUSIVE REMEDY OF THE CUSTOMER HEREUNDER (EXCEPT FOR ANY REMEDY OF REPAIR OR REPLACEMENT IF SO ELECTED BY FIORANO WITH RESPECT TO ANY BREACH OF THE LIMITED WARRANTY) SHALL BE LIMITED TO THE PRO-RATED AMOUNT OF FEES PAID BY CUSTOMER UNDER THIS AGREEMENT FOR THE PRODUCT, USING A TWELVE (12) MONTH STRAIGHT-LINE AMORTIZATION SCHEDULE STARTING ON THE COMMENCEMENT DATE. FURTHER, IF SUCH DAMAGES RESULT FROM CUSTOMER'S USE OF THE SOFTWARE OR SERVICES, SUCH LIABILITY SHALL BE LIMITED TO THE PRORATED AMOUNT OF FEES PAID FOR THE RELEVANT SOFTWARE OR SERVICES GIVING RISE TO THE LIABILITY TILL THE DATE WHEN SUCH LIABILITY AROSE, USING A TWELVE (12) MONTH STRAIGHT-LINE AMORTIZATION SCHEDULE STARTING ON THE COMMENCEMENT DATE. NOTWITHSTANDING ANYTHING IN THIS AGREEMENT, THE FOREGOING LIMITATIONS, EXCLUSIONS AND DISCLAIMERS SHALL APPLY TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, EVEN IF ANY REMEDY FAILS ITS ESSENTIAL PURPOSE.

The provisions of this Agreement allocate the risks between Fiorano and Customer. Fiorano’s pricing reflects this allocation of risk and the limitation of liability specified herein.

7. PAYMENT PROVISIONS.

(a) Invoicing. All fees shall be due and payable thirty (30) days from receipt of an invoice and shall be made without deductions based on any taxes or withholdings. Any amounts not paid within thirty (30) days will be subject to interest of the lower of the legal interest rate or one percent (1%) per month, which interest will be immediately due and payable.

(b) Payments. All payments made by Customer shall be in United States Dollars for purchases made in all countries except the United Kingdom, in which case the payments shall be made in British Pounds Sterling. Payments shall be directed to:

Fiorano Software, Inc.

718 University Ave.

Suite 212, Los Gatos, CA 95032

Attn: Accounts Receivable.

If the product is purchased outside the United States, payments may have to be made to an Affiliate as directed by Fiorano Software, Inc.

(c) Taxes. The fees listed in this Agreement or the applicable Order Form does not include Taxes. In addition to any other payments due under this Agreement, Customer agrees to pay, indemnify and hold Fiorano harmless from, any sales, use, excise, import or export, value added or similar tax or duty, and any other tax not based on Fiorano's net income, including penalties and interest and all government permit fees, license fees, customs fees and similar fees levied upon the delivery of the Software or other deliverables which Fiorano may incur in respect of this Agreement, and any costs associated with the collection or withholding of any of the foregoing items (the Taxes).

8. CONFIDENTIALITY.

(a) Confidential Information. Confidential Information shall refer to and include, without limitation, (i) the source and binary code of Products, and (ii) the business and technical information of either party, including but not limited to any information relating to product plans, designs, costs, product prices and names, finances, marketing plans, business opportunities, personnel, research, development or know-how;

(b) **Exclusions of Confidential Information.** Notwithstanding the foregoing, Confidential Information shall not include: (i) Information that is not marked confidential or otherwise expressly designated confidential prior to its disclosure, (ii) Information that is or becomes generally known or available by publication, commercial use or otherwise through no fault of the receiving party, (iii) Information that is known to the receiving party at the time of disclosure without violation of any confidentiality restriction and without any restriction on the receiving party's further use or disclosure; (iv) Information that is independently developed by the receiving party without use of the disclosing party's confidential information, or (v) Any Residuals arising out of this Agreement. Notwithstanding, any Residuals belonging to Source Code shall belong exclusively to Fiorano and Customer shall not have any right whatsoever to any Residuals relating to Source Code hereunder.

(c) **Use and Disclosure Restrictions.** During the term of this Agreement, each party shall refrain from using the other party's Confidential Information except as specifically permitted herein, and from disclosing such Confidential Information to any third party except to its employees and consultants as is reasonably required in connection with the exercise of its rights and obligations under this Agreement (and only subject to binding use and disclosure restrictions at least as protective as those set forth herein executed in writing by such employees).

(d) **Continuing Obligation.** The confidentiality obligation described in this section shall survive for three (3) years following any termination of this Agreement.Notwithstanding the foregoing, Fiorano shall have the right to disclose Customer's Confidential Information to the extent that it is required to be disclosed pursuant to any statutory or regulatory provision or court order, provided that Fiorano provides notice thereof to Customer, together with the statutory or regulatory provision, or court order, on which such disclosure is based, as soon as practicable prior to such disclosure so that Customer has the opportunity to obtain a protective order or take other protective measures as it may deem necessary with respect to such information.

9. MISCELLANEOUS.

(a) Export Administration. Customer agrees to comply fully with all applicable relevant export laws and regulations including without limitation, those of the United States (Export Laws) to assure that neither the Software nor any direct product thereof are (i) exported, directly or indirectly, in violation of Export Laws; or (ii) are intended to be used for any purposes prohibited by the Export Laws, including, without limitation, nuclear, chemical, or biological weapons proliferation.

(b) U. S. Government Customers. The Software is commercial items, as that term is defined at 48 C.F.R. 2.101 (OCT 1995), consisting of commercial computer software and commercial computer software documentation as such terms are used in 48 C.F.R. 12.212 (SEPT 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (JUNE 1995), all U.S. Government Customers acquire the Software with only those rights set forth herein.

(c) Notices. All notices under this Agreement shall be in writing and shall be deemed to have been given when mailed by first class mail five (5) days after deposit in the mail. Notices shall be sent to the addresses set forth at the beginning of this Agreement or such other address as either party may specify in writing.

(d) Force Majeure. Neither party shall be liable hereunder by reason of any failure or delay in the performance of its obligations hereunder (except for the payment of money) on account of strikes, shortages, riots, insurrection, fires, flood, storm, explosions, acts of God, war, governmental action, labor conditions, earthquakes, material shortages or any other cause which is beyond the reasonable control of such party.

(e) Assignment. Neither this Agreement nor any rights or obligations of Customer hereunder may be assigned by Customer in whole or in part without the prior written approval of Fiorano. For the avoidance of doubt, any reorganization, change in ownership or a sale of all or substantially all of Customer's assets shall be deemed to trigger an assignment. Fiorano's rights and obligations, in whole or in part, under this Agreement may be assigned by Fiorano.

(f) Waiver. The failure of either party to require performance by the other party of any provision hereof shall not affect the right to require such performance at any time thereafter; nor shall the waiver by either party of a breach of any provision hereof be taken or held to be a waiver of the provision itself.

(g) Severability. In the event that any provision of this Agreement shall be unenforceable or invalid under any applicable law or court decision, such unenforceability or invalidity shall not render this Agreement unenforceable or invalid as a whole and, in such event, any such provision shall be changed and interpreted so as to best accomplish the objectives of such unenforceable or intended provision within the limits of applicable law or applicable court decisions.

(h) Injunctive Relief. Notwithstanding any other provisions of this Agreement, a breach by Customer of the provisions of this Agreement regarding proprietary rights will cause Fiorano irreparable damage for which recovery of money damages would be inadequate, and that, in addition to any and all remedies available at law, Fiorano shall be entitled to seek timely injunctive relief to protect Fiorano's rights under this Agreement.

(i) Controlling Law and Jurisdiction. If this Software has been acquired in the United States, this Agreement shall be governed in all respects by the laws of the United States of America and the State of California as such laws are applied to agreements entered into and to be performed entirely within California between California residents. All disputes arising under this Agreement may be brought in Superior Court of the State of California in Santa Clara County or the United States District Court for the Northern District of California as permitted by law. If this Software has been acquired in any other jurisdiction, the laws of the Union of India shall apply and any disputes arising hereunder shall be subject to the jurisdiction of the Hon'ble City Civil Court, Bangalore, India. Customer hereby consents to personal jurisdiction of the above courts. The parties agree that the United Nations Convention on Contracts for the International Sale of Goods is specifically excluded from application to this Agreement.

(j) No Agency. Nothing contained herein shall be construed as creating any agency, partnership or other form of joint enterprise or liability between the parties.

(k) Headings. The section headings appearing in this Agreement are inserted only as a matter of convenience and in no way define, limit, construe or describe the scope or extent of such section or in any way affect such section.

(l) Counterparts. This Agreement may be executed simultaneously in two or more counterparts, each of which will be considered an original, but all of which together will constitute one and the same instrument.

(m) DISCLAIMER. THE SOFTWARE IS NOT SPECIFICALLY DEVELOPED OR LICENSED FOR USE IN ANY NUCLEAR, AVIATION, MASS TRANSIT OR MEDICAL APPLICATION OR IN ANY OTHER INHERENTLY DANGEROUS APPLICATIONS. CUSTOMER AGREES THAT FIORANO AND ITS SUPPLIERS SHALL NOT BE LIABLE FOR ANY CLAIMS OR DAMAGES ARISING FROM CUSTOMER'S USE OF THE SOFTWARE FOR SUCH APPLICATIONS. CUSTOMER AGREES TO INDEMNIFY AND HOLD FIORANO HARMLESS FROM ANY CLAIMS FOR LOSSES, COSTS, DAMAGES OR LIABILITY ARISING OUT OF OR IN CONNECTION WITH THE USE OF THE SOFTWARE IN SUCH APPLICATIONS.

(n) Customer Reference. Fiorano may refer to Customer as a customer in sales presentations, marketing vehicles and activities. Such activities may include, but are not limited to; a press release, a Customer user story completed by Fiorano upon implementation of the Software, use by Fiorano of Customer's name, logo and other marks, together with a reasonable number of technical or executive level Customer reference calls for Fiorano.

(o) Entire Agreement. This Agreement, together with any exhibits, completely and exclusively states the agreement of the parties. In the event of any conflict between the terms of this Agreement and any exhibit hereto, the terms of this Agreement shall control. In the event of any conflict between the terms of this Agreement and any purchase order or Order Form, this Agreement will control, and any pre-printed terms on Customer's purchase order or equivalent document will be of no effect. This Agreement supersedes, and its terms govern, all prior proposals, agreements or other communications between the parties, oral or written, regarding the subject matter of this Agreement. This Agreement shall not be modified except by a subsequently dated written amendment signed by the parties, and shall prevail over any conflicting pre-printed terms on a Customer purchase order or other document purporting to supplement the provisions hereof.

Exhibit A

Fiorano Product List

Each of the individual items below is a separate Fiorano product (the Product). The Products in this list collectively constitute the Software. Fiorano reserves the right to modify this list at any time in its sole discretion. In particular, Product versions might change from time to time without notice.

1. Fiorano SOA Enterprise Server
2. Fiorano ESB Server
3. FioranoMQ Server Peer
4. Fiorano Peer Server
5. Fiorano SOA Tools
6. Fiorano Mapper Tool
7. Fiorano Database Business Component
8. Fiorano HTTP Business Component
9. Fiorano SMTP Business Component
10. Fiorano FTP Business Component
11. Fiorano File Business Component
12. Fiorano MOM Business Components (MQSeries, MSMQ, JMS)

NOTE: Other business components may be added to or removed from this list from time to time at Fiorano's sole discretion.

Exhibit B

EXCLUDED COMPONENTS

(a) Any third party or open source library included within the Software

Exhibit C

Licensing Restrictions. The Software licensed hereunder is subject to the following licensing restrictions.

The parties understand that the modules of the Software are licensed as noted in this section. The term Target System means any computer system containing one or more Processors based upon any architecture, running any operating system, excluding computers running IBM MV-S, OS/390 and related mainframe operating systems. The Term Processor means a computation hardware unit such as a Microprocessor that serves as the main arithmetic and logic unit of a computer. A Processor might consist of multiple Cores, in which case licenses shall have to be purchased on a per-Core basis. A Target System may have one or more Processors, each of which may have one or more Cores. In the sections below, Cores may replace Processors as applicable.

(a) If the Software is Fiorano ESB Enterprise Server, FioranoMQ Peer, Fiorano SOA 2007 server or FioranoMQ Server (JMS), then the Software is licensed on a per Processor basis on a single Target System, where the total number of Processors on the Target System may not exceed the total number of Processors licensed, with the additional restriction that only a single instance of the Fiorano ESB Enterprise Server may run on a single Target System and that a separate license must be purchased for each instance of the Fiorano ESB Enterprise Server, Fiorano ESB Peer Server or FioranoMQ Server (JMS) Server for each Processor;

(b) If the Software is Fiorano SOA 2007 Tools or Fiorano Mapper Tool 2007, or any Fiorano Test and/or Development license, then the Software is licensed on a per-named-user basis, where the total number of named users may not exceed the total number of named users licensed;

(c) If the Software is a Fiorano Business Component of any kind (including but not limited to Fiorano HTTP, File, SMTP, File, Database, and other Business Components, etc.), then the Software is licensed on the basis of the number of CPUs of the Target System on which the FioranoMQ Peer (to which the Business Component connects runs). A separate license needs to be purchased for each CPU of each Target System of each FioranoMQ Peer instance to which any Business Component connects.

Evaluations. Licenses used for evaluation cannot be used for any purposes other than an evaluation of the product. Existing customers must purchase new licenses to use additional copies of any Product and may not use evaluation keys in any form. All evaluation keys are restricted to 45-days and extensions need to be applied for explicitly. Any misuse of evaluation keys shall be subject to a charge of 125% (one hundred and twenty-five percent) of the license fee plus 20% support.

Copyright (c) 1999-2008, Fiorano Software Technologies Pvt. Ltd.,

Copyright (c) 2008-2009, Fiorano Software Pty. Ltd.

All rights reserved.

This software is the confidential and proprietary information of Fiorano Software ("Confidential Information"). You shall not disclose such ("Confidential Information") and shall use it only in accordance with the terms of the license agreement enclosed with this product or entered into with Fiorano.

Contents

Chapter 1: Introduction ... 23

Messaging Fundamentals ... 23
- JMS Provider ... 24
- Loosely coupled nature of Messaging Systems ... 24
- Reliable Delivery of Messages ... 24
- Messaging Domains ... 24
- Mode of Consumption of Messages ... 24
- Administered Objects ... 25
- Sessions ... 25
- JMS Message ... 25

Salient features of FioranoMQ ... 26
- High Availability ... 26
- Clustering ... 26
- XA Support ... 26
- Scalability ... 26
- Application Server Integration ... 26
- Native runtime support ... 27
- Security ... 27
- Durable Connections ... 27
- Large Message Support ... 27
- Hierarchical Topics ... 27
- HTTP Support ... 27
- Logging Facilities ... 28
- Message Snooping ... 28
- Dead Message Queue ... 28
- Encryption, Compression Support ... 28

Samples ... 28

Chapter 2: Configuration Concepts ... 29

Fiorano Component Model ... 29
Deployment Profile ... 31
Default Profiles ... 32
Configuration Tools ... 32

Chapter 3: Connection Management ... 34

Socket Acceptors ... 34
- Port Number ... 34

Protocol ... 34
Thread Management ... 35
Security Parameters ... 35
Configuration ... 35
Connection Factory ... 35
Obtaining a Connection Factory Instance ... 36
JNDI Lookup ... 36
Creating a new instance ... 36
Lookup ... 37
JMX ... 37
RMI Connector ... 37
JMS Connector ... 37
Pinging ... 37
When to Enable Pinging ... 37
Salient Features ... 38

Chapter 4: HTTP Support ... 39

Client Side Changes ... 39
Using Proxies ... 39
Proxy Authentication ... 39
Tunneling through Firewalls ... 40
Tunneling through SOCKS Proxy Server ... 40
Enabling JMS Applets to Tunnel through SOCKS Proxy Server ... 41
Additional Notes on SOCKS ... 41
HTTP Pinging ... 41

Chapter 5: FioranoMQ Security ... 42

User Identification and Authentication ... 43
Data Protection ... 43
Authentication Based on Digital Certificates ... 44
Security Realms ... 44
FioranoMQ User Management ... 45
Access Control Management ... 45
Default Realm ... 46
NT Realm ... 46
Salient Features ... 46
Limitations ... 46
Troubleshooting ... 47
RDBMS Realm ... 47
LDAP Realm ... 48
Configuring the LDAP Security Realm ... 48
Miscellaneous Features ... 49
XML Realm ... 49

Caching Realm 50
FioranoMQ Security - Salient Features & Advantages 50
Design Advantages 50
Effective Protection of JMS Destinations 50
Centralized Control 50
Destination-based Security 51
Authorization and Access Control 51
Default Users, Groups and ACLs 52

Chapter 6: FioranoMQ Data Stores 53

Storage type 53
File-based store versus JDBC-compliant RDBMS store 53
Default Destinations for sample applications 54
Creating a default database 54
Clearing a database 55

Chapter 7: Managing Administrated Objects 56

Naming Services 56
File 56
XML 57
LDAP 57
RDBMS 57
Cache 57
Salient Features 58

Chapter 8: Message Expiry 59

Point of Checking of Message Expiry 59
On Detection of an Expired Message 59
Dead Message Queue 59
DMQ Configuration 60
Selectively disabling DMQ for a message 60
Message Expired Notifications 60
Configuration 60
Salient Features 61

Chapter 9: Snooper 62

Snooper Configuration 62
Working of Snooper 63
Security Settings 63
Miscellaneous Features 63

Chapter 10: Durable Connections 64

Overview 64
Working of Durable Connection 65
Producer on a Durable Connection 65
Consumer on a Durable Connection 65
Advantages 65
- Network reliability 65
- Store and Forward capabilities 66
- Transparent reconnection code 66
- Message browsing of persisted messages 66
- No vendor lock in 66

Enabling Durable Connections Support 66
Client side Message Cache 66
Serverless Environment 67
- Sample Application 68

Relationship with Revalidate 69
Relationship with CSP 69
Constraints in Durable Connections 70

Chapter 11: Hierarchical Topics 71

Need For Hierarchical Name Spaces 71
Name Space Notation 71
Creating Hierarchical Topics 71
- Case Insensitive 71
- Spaces in Names 72
- Empty String 72
- Unlimited Length of Topic Names 72
- Unlimited Depth of Topic Hierarchy 72
- Wild Card Support 72

Dynamic Creation of Topics in Hierarchy 72
Looking up Hierarchical Topics 72
Publishing on Node(s) in Topic Hierarchy 73
Subscribing to Node(s) in Topic Hierarchy 73
- Template Characters Used in Subscription 73

Deleting a Hierarchical Topic 75
Publish/Subscribe across Servers 75
Security Considerations on Hierarchical Topics 75
Limitations 76

Chapter 12: Message Encryption 77

Base Implementation 77
Message Encryption Characteristics 78

Chapter 13: Message Compression 79

Base Implementation 79
Message Compression Characteristics 80

Chapter 14: FioranoMQ Clustering 81

Common problems of Real-world Systems 81
Client unable to connect 81
Connection to the server is lost 81
The server runs out of resources 81
The server goes down altogether 82
FioranoMQ: The solution 82
Automatic Fail over protection 82
Transparency and code portability 82
Configurability 82
Admin system 83
Connection to the server is lost 83
Server runs out of resources 84
Server's connection to a client is lost 84
Server-to-server communication 84
Scalability 84
Clustering Components 85
Dispatcher 85
Preferred Server 86
Configuration Parameters 87
Repeater 87
Salient Features 89
No changes in the client application 89
Robustness in handling network failures 89
Subscription Mode & Choice of Selectors 90
Request/Reply Across Repeater 90
Dynamic Replication Links 90
Repeater with Load Balancing 90
Repeater Link 90
Connection Info 91
Link Topic Info 91
Configuration Parameters 91
Wild Character Support 92
Dynamic Link Propagation 92
Refresh the Repeater 93
Bridge 93
Bridge Architecture 93
Forwarding Messages to Remote Queues 94
Bridge Features 95
Bridge Configuration 96

Property Name Description 96
Link Properties 97
SourceServer 97
TargetServer 97
Connection Info Properties 97

Chapter 15: Large Message Support 99

Salient Features 99
- Reliable transfer of large messages 99
- No increase in cache/JVM heap size required 99
- The Large message transfer is not restricted to any queue or topic 99
- Resume functionality at both sender and receiver end 99
- Minimal changes in the application code 99

Using Fiorano LMS to transfer large files 100
- Message Creation 100
- Starting the message transfer 100
- Tracking the message transfer 100
- Resuming the message transfer 100

Salient Features 101
- Consumer Discovery 101
- Fragment size 101
- Window size 101
- Sequencing 101
- Handling Duplication 102
- Handling lost fragments 102

Optimizing large message transfer 102
- Fragment size 102
- Window size 102
- Status message frequency 102

Chapter 16: High Availability 104

FioranoMQ's HA - An overview 104
HA Components 105
- Backup Server 105
- Server States 105
- Intra-Enterprise Server Communication 106
- Common Persistent Message Store 106
- Common Admin and Security 106
- Gateway Machine 106

FioranoMQ HA Salient Features 107
- Shared and Replication database 107
- Application Fail over 107
- Data Store Consistency maintained between server switches 107

Expensive HA Hardware Not Required 107
Implementing a Cluster 107
HA Example Scenario 108
State - 1 (Normal Operation State) 108
State - 2 (Active Server goes down) 109
State - 3 (Backup Server resumes operations) 109
Limitations of HA 110

Chapter 17: Distributed Transactions 111

Introduction 111
Use Case 112
Transactions and DTP System 113
Components of a Distributed Transaction 113
FioranoMQ as a Distributed Transaction Resource Manager 114
Transactions with J2EE 115
FioranoMQ XA Implementation Notes 116
LimitatiHons of XA Implementation of FioranoMQ 117

Chapter 18: FioranoMQ Content Based Routing 121

Introduction 121
FioranoMQ Content Based Routing 121
Using FioranoMQ Content Based Routing 122
Setting up the FioranoMQ Server for CBR 122
FioranoMQ CBR XPath Support 123
Publishing XML Messages 124
Subscribing to XML Messages 125
CreateDurableSubscriber 126
CreateSubscriber 126
Handling Massive Number of Subscribers 129
XML Support 130
FioranoMQ Content - Based Message Selector Language 131
General Form 131
Subset of Supported XPath Queries 132
Identifiers 133
Operators 133
Literals 135
Example XMLs 135
Elements Only XML 135
Attributes Only 136
Elements and Attributes XML 137
Limitations of FioranoMQ Content Based Routing 138

Chapter 1: Introduction

An important requirement for business enterprises is the exchange of critical business data and events throughout the enterprise. Messaging is a mechanism that provides communication between software applications or objects in a distributed system. JMS provides these enterprises with a foundation of messaging.

Fiorano's messaging solution is based on JMS 1.1 standards for enterprise messaging.

FioranoMQ is a high-performance, stable, secure, and pure Java implementation of JMS. The development time of the applications which require a messaging infrastructure is greatly reduced by using Fiorano's messaging solution.

Automatic store-and-forward capability across multiple servers ensures high scalability, high availability, high performance and guaranteed message delivery across the faulty networks.

FioranoMQ includes distributed transaction support ensuring high levels of consistency and reliability. This aids the process of developing and deploying Internet, intranet and extranet applications.

FioranoMQ is a JMX-enabled server thereby making it easy for administrators to manage and monitor it.

It provides native runtime libraries written in C, C++ and C# for all major platforms. These native runtime libraries allow non-java clients to talk directly to the java server and exchange information with other JMS-compliant clients.

The security implementation includes integrated JSSE support. The Java Secure Socket Extension (JSSE) enables secure Internet communications. It implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols. Developers can thus provide for secure channels for data transfer between a client and a server.

Messaging Fundamentals

This section explains you the concept of message fundamentals. The types of message explained in this section are:

- JMS Provider
- Loosely coupled nature of Messaging Systems
- Reliable Delivery of Messages
- Messaging Domains
- Mode of Consumption of Messages
- Administered Objects
- Sessions
- JMS Message

JMS Provider

In JMS parlance, a provider implements JMS interfaces defined by the JMS specifications for a messaging product. It also provides some administrative and control functionality.

In a messaging system, each client connects to the messaging provider. Each message is addressed to a particular destination, maintained by the provider. Client applications have the capability of producing messages, intended for a destination and consuming messages, from the destination. The provider takes the responsibility for the routing and persistence of messages.

Loosely coupled nature of Messaging Systems

Senders and receivers are anonymous to each other since each client connects to the messaging provider. The producer and the consumer require the name of the destination maintained by the provider to which the producer sends the messages and from which the consumer consumes messages. This loosely coupled nature of messaging systems is their biggest advantage. It allows an enterprise to continue to operate even when parts of it are disabled, makes them scalable, and enables sharing of widely distributed resources.

Reliable Delivery of Messages

The sender and the receiver do not have to be available at the same time. The provider persist these messages and guarantees their delivery when the client comes up.

Messaging Domains

There are two kinds of messaging domains:

- Point-to-point (PTP)
- Publish-subscribe(Pub/Sub)

In PTP domains, messages are sent to a particular queue destination. A client application is delivered these messages from the queue by the provider. Many senders can send messages to the same queue. A particular message is intended for only one receiver.

In Pub/Sub domains, the same message can be broadcast to many subscribers. A message is published on a topic destination. The provider further delivers the message to one or more subscribers subscribed on the topic in question.

FioranoMQ implements both the domains and also the unified domain concept that has been introduced in JMS 1.1.

Mode of Consumption of Messages

A message can be consumed in two ways: either synchronously or asynchronously.

In the synchronous mode, the client application can request for the next message. There are variations of how the client receives the message by either blocking infinitely or for a finite timeout period.

In the asynchronous mode, the client application defines a message listener. Whenever a message arrives for that destination, the provider delivers the message to the subscriber by invoking the OnMessage method of the listener. No blocking is involved.

Administered Objects

JMS providers can differ in their implementation of the JMS specification and in the installation and administration of the messaging system. For JMS clients to be portable, the proprietary parts are encapsulated in JMS administered objects and are created by the provider's administrator. These administered objects are ready for use by clients via JMS interfaces.

They are stored in a JNDI namespace.

There are two types of JMS administered objects:

- ConnectionFactory - the object a client uses to create a connection with a provider
- Destination - the object a client uses to specify the destination of messages it is sending to and from which a client receives messages.

Sessions

A session is a single-threaded context for producing and consuming messages. It can create and service multiple message producers and consumers.

A session may be specified as transacted or non-transacted.

Each transacted session supports a single series of transactions and treats them as an atomic unit. The content of a transaction are the messages that have been produced and consumed within a transaction. A transaction is completed using a commit method which indicates that message processing can occur or by using a rollback method wherein the messages in that transaction are not processed.

A non-transacted session acknowledges the receipt of a message in three modes as per the JMS 1.1 specification: AUTO_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE, and DUPS_OK. The DUPS_OK is used in applications wherein duplicate delivery of messages is tolerated. So the client application "lazily" acknowledges a message. This method is most efficient in terms of usage of resources.

JMS Message

A JMS message is composed of a header, properties - a facility for adding optional header fields for a message, and a body. There are five kinds of messages: StreamMessage, MapMessage, TextMessage, BytesMessage, and ObjectMessage. FioranoMQ extends TextMessage by providing an additional message type: XMLMessage

Salient features of FioranoMQ

High Availability

Systems like some financial systems that require near-zero downtime, must provide high availability solutions. FioranoMQ HA deployment allows JMS clients to transparently switch over to a secondary MQ server on failure of the primary server.

Client applications are provided with capabilities like automatic re-connection to the backup server and store and forward capability. In case of a fault, all the information that was persisted in the primary server is made available to the applications when they connect to the backup server. This provides applications with automatic fault-tolerance capabilities and allows them to focus on the business logic. This makes loss of connection issues with the JMS provider transparent to the application.

Clustering

A server cluster consists of multiple server instances running concurrently to provide increased scalability and reliability. To clients it appears as if one FioranoMQ server instance is running. Clustering enables clients connected on different FioranoMQ servers to exchange information without each client having to connect to each server. FioranoMQ provides clustering support with the help of Dispatcher, Repeater and Bridge components.

Fiorano's load balancing architecture involves the use of a Dispatcher-enabled server, to route incoming client connections to the least loaded server in a cluster. The dispatcher component is connected to multiple FioranoMQ servers. All these servers become part of the cluster that is served by the dispatcher. The repeater and bridge components are used for server-to-server communication over topics and queues respectively.

XA Support

Many real world applications require transactions involving multiple resource managers. These transactions are known as distributed transactions or global transactions. Implementation of distributed transactions involves following the JTA standards. FioranoMQ supports both local and global transactions through the same context. If a global transaction is active, all activities performed become part of this transaction; otherwise they operate locally as in a normal JMS transaction.

Scalability

The load balancing and failover protection architecture allows high scalability in terms of the number of concurrent client connections that a FioranoMQ server can service.

Application Server Integration

FioranoMQ integrates seamlessly with application servers. Some of the Application Servers that FioranoMQ integrates with are: JBoss, WebLogic, Orion, and iPlanet.

Native runtime support

FioranoMQ provides native runtime libraries written in C, C++ and C# for all major platforms. These native runtime libraries allow non-java clients to talk directly to the java server and exchange information with other JMS-compliant clients.

Security

The security implementation includes integrated JSSE support. The Java Secure Socket Extension (JSSE) enables secure Internet communications. It implements a Java version of SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols and includes functionality for data encryption, server authentication, message integrity, and optional client authentication. Developers can thus provide for secure channels for data transfer between a client and a server running any application protocol over TCP/IP.

Durable Connections

Durable Connection support provides client applications with a fault-tolerance connection mechanism. If an application creates a durable connection, it need not worry about re-connecting back to the server in case of some fault. This is automatically handled by FioranoMQ's runtime library. If any message is sent during the disconnected phase, it is stored in a local repository in the client machine.

Large Message Support

Support is provided for JMS applications to transfer large messages (or files). The transfer could employ either the point-to-point model or the publish/subscribe model. Fiorano's implementation takes care of resuming data transfer from the point of failure. FioranoMQ enables applications to transfer large messages by shielding them from memory usage issues in the client machine as well as the server machine.

Hierarchical Topics

An organization organizes its data depending on its content. JMS achieves this by associating messages with specific destinations. These destinations are not related to each other. An organization may need to impose a hierarchical structure on its data. To provide a logical correlation between topics, the concept of hierarchical topics has been introduced. A topic can have a child topic thus developing into a hierarchical structure of topics.

HTTP Support

In order to allow enterprise users secure access outside the company firewall, FioranoMQ provides Hypertext Transfer Protocol (HTTP) over Secure Sockets Layer (SSL).

Logging Facilities

FioranoMQ incorporates tracing and logging facilities for easy detection of errors in the messaging system. The FioranoMQ Administrator can dynamically set different tracing levels for each individual FioranoMQ component.

Message Snooping

Administrators can view the messages that have been published on both topics and queues. The ability to snoop messages facilitates the administration, management, testing, and debugging of JMS applications.

Dead Message Queue

A message can be associated with a timeout period within which it should be received. A dead message queue stores messages whose timeout period has expired or are undeliverable. This enables the administrator to get more information on the status of delivery of a message.

Encryption, Compression Support

Encryption and Compression support can be selectively applied for a message or for all messages associated with a destination. The default implementation of encryption is based on DES. The default implementation of compression is based on Zlib implementation.

Note: Message browsing is another feature that is provided.

Samples

FioranoMQ comes bundled with sample applications that exhibit its features. Experimenting with these sample applications gives a hands-on perspective of the product.

Chapter 2: Configuration Concepts

This section provides an insight into the configuration module used by FioranoMQ.This section provides introduction for the following:

FioranoMQ's componentization model

- Deployment Profile
- Configuration Modes
- Off-line
- On-line
- JMX
- API's

This chapter explains the component model followed by FioranoMQ and then introduces the reader to the concept of "Deployment Profile" along with explaining the pre-created profiles bundled with FioranoMQ Installer. The user is then introduced to various options to configure the server.

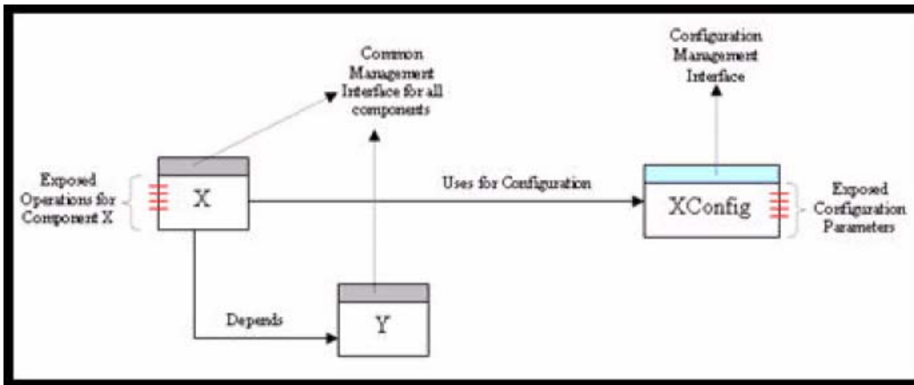
Fiorano Component Model

FioranoMQ server implements a component model for various internal modules. These components can be clubbed together to form a deployment profile. Each deployment profile can be separately configured both in offline and online mode.

FioranoMQ Server comprises of a number of components that implement a well-defined functionality independently.

A component

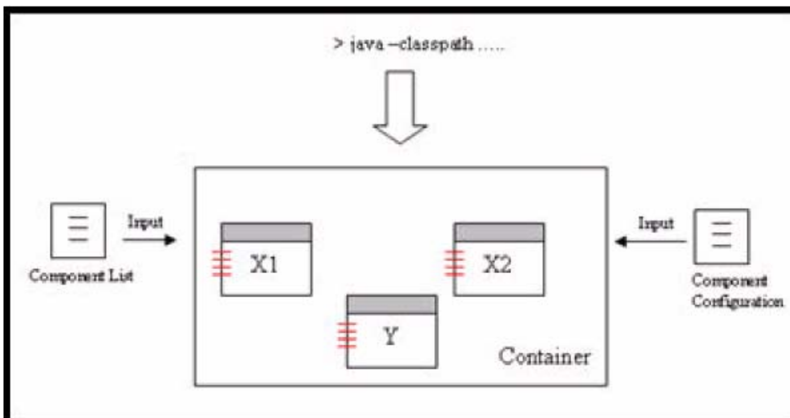
- Has a well-defined interface through which its life cycle can be controlled.
- Is associated with a unique configuration object that defines its configuration needs.
- May expose configuration attributes or operations to the external world via JMX.
- Defines dependencies on other components.



A component is not a stand-alone executable application but can be hosted only within a Container. A container:

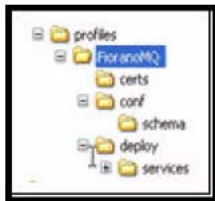
- Is an Executable Java Program
- Takes a list of components as input.
- Resolves Component dependencies and launches components in the correct order.
- Provides access to Component's Configuration on its launch.
- Encapsulates a JMX MBean Server to which all deployed components are bound.

Allows invocation of exposed operations and changes in exposed configuration parameters for a component.



Deployment Profile

As explained above, a component just provides a set of services without knowing much about its surroundings. It is not aware of the "application" that it is a part of. The container too is unaware of the content of the "application". The container just looks at a group of components working together in perfect harmony. The "application" is defined by the list of components (deployment.lst) that is taken as an input by the container. By modifying this list, one can modify the behavior of the application hosted in the container. This deployment list along with the persisted configuration for components essentially makes up a "deployment profile". In other words, a deployment profile consists of a list of files (meant for deployment & configuration) organized in a pre-defined directory structure. A typical profile structure (on which the server hasn't been run) is as shown in the figure below. The table below describes these directories.



Directory	Description
certs	Certificates used if the server is running on SSL
conf	Database Configuration Files (*.properties & *.cfg) Configs.xml [Persisted configuration of components.]
deploy	*.lst files – These files contain the list of components that are to be deployed in the container when using this profile.
deploy/services	XML Files for various components defining their dependencies & relative object Names.
run	This directory will get created once a profile is run for the first time. This is the default file-based database storage location for a profile.

In other words, FioranoMQ Server is an "application" that runs within the Container. FioranoMQ Server consists of a number of components that provide various functionalities like pubsub, ptp, admin etc. Modifying the deployment profile can alter the behavior of the server. For example it is possible to run a bridge or repeater component along with the FioranoMQ Server within the same JVM (NOTE - Prior to 8.x Bridge & Repeater used to run in a separate JVM). This has been made possible by componentization of bridge and repeater. One can turn on or off FioranoMQ features like XA by including or removing the appropriate component from the deployment profile.

Each Profile can be logically thought of as a separate and independent "work-area". This work area besides hosting the configuration and deployment information also serves as the default location for File Based Data stores and generated Logs for the FioranoMQ server. The data store and logs are generated in the "run" folder that is created on the fly when the server is launched for the first time.

Default Profiles

Default FioranoMQ Server installation comes with some pre-created profiles that are configured to demonstrate certain Server Functionalities. These profiles are located in "profiles" directory of the server installation and are summarized in the table below.

Profile	Description
FioranoMQ	Default FioranoMQ profile
FioranoMQ_HA_rpl/HAPrimary FioranoMQ_HA_rpl/HASecondary	Pre-configured profile for running primary fmq server with HA (replication) enabled Pre-configured profile for running secondary fmq server with HA (replication) enabled
FioranoMQ_HA_shared/HAPrimary FioranoMQ_HA_shared/HASecondary	Pre-configured profile for running primary fmq server with HA (shared) enabled Pre-configured profile for running secondary fmq server with HA (shared) enabled
FioranoMQ_XA	Pre-configured profile for running fmq server with XA enabled
StandAloneBridge	Pre-configured profile for running bridge on an independent JVM
StandAloneRepeater	Pre-configured profile for running repeater on an independent JVM

From the list of available profiles, the Container by default runs over "FioranoMQ" profile. In order to point it to some other profile, specify the profile name along with "-fmq.profile" parameter in the command line when launching the container.

For example, in order to use "FioranoMQ_XA" profile, launch the container using the command line: fmq.bat -fmq.profile FioranoMQ_XA

Configuration Tools

FioranoMQ can be configured in the following ways:

- FioranoMQ provides a graphical tool called Studio built over NetBeans platform to configure and manage one or more FioranoMQ Servers.
- Configuration information can be accessed & modified through Studio even if the server is not available. This mode of editing configuration is described as "Offline Configuration". The tool for this is called Profile Manager.
- Studio allows a user to manage a running instance of FioranoMQ server. This mode of editing configuration is described as "Online Configuration".
- FioranoMQ provides comprehensive support for JMX. Any standard JMX-compliant tool can be used to administer FioranoMQ server. Fiorano ships a JMX-compliant administration tool called Fiorano JMX explorer along with FioranoMQ.

- Administration using FioranoMQ proprietary administration API.

Chapter 3: Connection Management

FioranoMQ server modules include a transport layer that interfaces with the underlying network protocols to accept incoming connections and the requests coming on them. This layer is responsible for reading the requests sent by applications over the established communication.

Some of the important responsibilities of the transport layer are:

- Listening for incoming client connections on specified protocol (HTTP/TCP etc).
- Thread Management.
- Parsing incoming data and passing on the incoming request to the core FioranoMQ Services.
- Detecting loss of connectivity.

Socket Acceptors

Socket Acceptor basically represents the input ports on which the server listens for incoming connections. Each socket acceptor is associated with a physical port number, a transport protocol, a connection manager and optional security parameters.

These are explained as follows:

Port Number

This refers to the physical TCP/IP port on which the server listens for incoming connections. Once a connection is established, the socket acceptor is used for handing all requests coming from the client application. This includes JMS, Admin, Lookup & internal asynchronous requests that come from FioranoMQ runtime library.

Note: Since a port cannot be shared between two applications running on the same machine, the port number used by the FioranoMQ server is unique to a server instance. If required to run two instances of FioranoMQ server, on a machine, it is required, that both the servers listen on different ports. JMX Requests reach the server via the plugged in JMX Connector. This is independent of the SocketAcceptor being used.

Protocol

The protocol refers to the physical transport that a client is required to follow in order to connect to the server. By default the server is configured to use TCP. Other possible options are HTTP. Additionally, SSL can also be enabled over both TCP as well as HTTP.

Thread Management

FioranoMQ server offers different thread management schemes that differ in their policies of handling new connections. The default thread management scheme associated with a socket acceptor is configured to create a new thread for each connection (Socket) connected with the server. Other schemes allow configuring a thread pool that would service requests coming from all the connections. This ensures the number of threads in the server remains fixed and hence the server's resource requirement becomes independent of the number of connections.

Security Parameters

The socket acceptor can be configured to enable security through SSL. This can be done both on TCP protocol as well as HTTP protocol. FioranoMQ server provides implementation of SSL over TCP using the Phaos Toolkit and Sun's JSSE. By default SSL is not enabled.

Configuration

By default, FioranoMQ server is configured with one Socket Acceptor. This socket acceptor is configured to listen on port 1856 and uses the TCP protocol. The FioranoMQ administrator has the following privileges with respect to socket acceptors

- Can edit the default socket acceptor configuration in any manner.
- Create additional socket acceptor(s) with any configuration.

Note: An additional SocketAcceptor in the server opens another port (as configured) for communication over the specified protocol.

Connection Factory

As per JMS specifications, an application uses a connection factory instance to connect to the server. The connection factory instance encapsulates all the parameters (like URL, protocol etc) required to connect to the server. These parameters are by default configured to use the default socket acceptor settings and hence requires modifications if the server uses a socket acceptor with non-default configuration.

The server creates the default connection factories when it is launched for the first time. These connection factories are automatically created based on the configuration of the socket acceptor being used.

Note: In case multiple Socket Acceptors are being used, the default connection factories are created by using the parameters of any one. The server can be forced to re-create default connection factories after resetting the database and starting the server.

Connection Factory Configuration Parameters

Parameter Name	Description	Default Value
ConnectURL	The server URL in the following format Error! Hyperlink reference not valid.> **Note** – The protocol to be used is not part of the URL.	http://localhost:1856
BackupURLs	Semi-colon separated list of URLs that should be tried when creating connection (or when revalidating) if connection with the server specified in connectURL cannot be established (or fails).	
IsForLPC		
isConnectURLUpdationAllowed	If set to true, the connection factory will have the IP address & port of the server through which it is looked up. This flag is useful for machines where IP address or port changes. **Note** – This flag is turned on for default connection factories.	False.
ConnectionClientID	If not null, this represents a client ID that would automatically be set on a connection created through this connection factory.	Null
PingDisabled	If set to true, a connection created through this connection factory will not be pinged even if pinging is turned on at the server.	False

Obtaining a Connection Factory Instance

A connection factory is a stateless object that just encapsulates information on how to connect to the server. An instance of the same can be obtained in either of the following ways.

JNDI Lookup

A connection factory instance is a serializable object that can be stored and later looked up through any JNDI-compliant directory server. For convenience, FioranoMQ provides the JNDI interface to lookup all admin objects.

Creating a new instance

An application can simply create a new instance of connection factory and use the same after setting various configurable parameters as desired.

Lookup

Using FioranoMQ, applications can lookup various objects (like destinations & connection factories) using the JNDI interface. As described earlier, a single socket acceptor can service lookup requests besides the normal JMS requests. Therefore in order to send the request to the server, its connection parameters have to be specified as environment variables to JNDI. Server URL, Transport protocol and security parameters can be specified in the environment passed to the JNDI layer when looking up an object from the server.

JMX

FioranoMQ 8.0 and upward provide extensive support for JMX. This allows any third party JMX-compliant applications to connect to the server remotely and access/modify the configuration at runtime. This requires a JMX Connector to be plugged in that services incoming JMX requests. FioranoMQ provides the following two options for JMX Connectors

RMI Connector

This is the default connector shipped with FioranoMQ server. It uses RMI as the underlying transport protocol to establish communication between a JMX-compliant application and the server. This connector uses a dedicated socket for accepting connections and servicing requests coming on them. The Connector by default listens on port 1858 but can be configured easily to listen on any other port.

JMS Connector

This connector uses the underlying JMS Bus for establishing communication between the JMX Application and FioranoMQ Server. This takes all connection parameters as configurable parameters. By default it is configured to connect to an FioranoMQ Server running over the default socket acceptor configuration. In case the socket acceptor configuration is modified, appropriate changes have to be made in JMS Connector's configuration as well.

Pinging

Another important responsibility of the transport layer is to detect loss of connectivity (between the application and the server) and do the necessary cleanup in the server. This feature known as Pinging, when enabled (by default it is turned off) instructs the FioranoMQ library to send ping packets periodically (automatically) over the connected sockets to the FioranoMQ server. The server on its part monitors ping requests on all connections and if it doesn't receive ping packets for any connection within a configurable timeout, it assumes the connection to be dead and hence closes it.

When to Enable Pinging

In some operating systems, the absence of any activity over a client socket for some time leads to its forceful closure by the OS. This can be avoided by enabling Pinging.

For an application that is not actively sending any requests to the server (a subscriber application for instance), a network failure might remain undetected. This can be avoided by turning on Pinging. With Pinging on, the application is notified, of the problem, within the configurable timeout.

Salient Features

Configuration Parameter for Pinging includes Ping Timeout Interval. This parameter specifies the time within which the client application is to be notified of a problem (with connectivity) if it exists. By default, this parameter is set to 4 minutes or 240,000 ms. and the minimum value allowed for this parameter is 30,000 ms.

Pinging is automatically turned on, when a Socket Acceptor is using HTTP protocol.

Since a connectivity problem is detected asynchronously for the application, the only way to inform it about the error is through an exception listener set on the connection as per JMS Specifications.

Note - In other words, it is mandatory to set an exception listener on the connection if the application wants to be notified of connectivity problems detected asynchronously through Pinging.

Chapter 4: HTTP Support

FioranoMQ supports pure HTTP as the transport protocol between JMS clients and MQ server. This allows the JMS communication to seamlessly flow through the corporate firewalls/proxies. This feature implementation makes sure to insulate the protocol layer implementation from the JMS application development. The client side environment is responsible for determining the protocol implementation to be used for communication. From an end user's perspective, all the protocols namely - TCP, Secure TCP (Phaos), Secure TCP (JSSE), HTTP, Secure HTTP (Phaos), Secure HTTP (JSSE) behave in a similar way, the selection being made solely on the basis of the configured environment of the application developer. Both synchronous and asynchronous communications are available, regardless of the protocol choice.

To deploy on the internet, usually HTTP is used. If the FioranoMQ server has to directly process messages received from the clients over the Internet, it must be deployed as if it were a web-server. The HTTP support in FioranoMQ provides the necessary features, which makes it function as a web server that can handle HTTP requests. Thus, you can establish a direct connection between client and server using HTTP Tunneling as the protocol.

Client Side Changes

When switching the protocol in the server from TCP to HTTP, the following changes are required at the application level.

Pass additional parameters as JNDI environment, if looking up from FioranoMQ. If jndi.properties file is used to specify these parameters, application code is not required to be modified. Use HTTP Enabled Connection factory.

Include HTTPClient.zip in classpath if not already included. In other words, application code may not change at all.

Using Proxies

The HTTP support of FioranoMQ provides seamless communication through proxy servers. FioranoMQ clients can connect to the FioranoMQ server through most popular proxy servers such as, Microsoft ISA server, Netscape proxy, Wingate, and WinProxy. FioranoMQ Client libraries allow developers to set Proxy Address and port in the client applications and also as a Java VM Properties.

Proxy Authentication

FioranoMQ supports both "Basic" and "Digest" authentication for communication through proxies. Various Proxy Authentication parameters such as the Authentication Realm username and password can be specified from the client application through environment variables.

Authentication is required only once within the instance of a VM. FioranoMQ caches this information and uses it for all the other connections.

Tunneling through Firewalls

This section discusses how JMS Clients interoperate in networks where firewalls are present. FioranoMQ allows enterprise clients to extend beyond the firewall of your corporation by providing both Http tunneling and tunneling through SOCKS enabled Proxy Servers. FioranoMQ provides Tunneling support for clients with all the JMS functionalities.

Tunneling through SOCKS Proxy Server

Tunneling through client as well as server side firewalls can be achieved through SOCKS Proxy Server. The SOCKS protocol is an open Internet standard for performing network proxying at the transport layer. SOCKS create a proxy, which serves as a data channel between a TCP or UDP based client and server. The proxy between the client and server, created by SOCKS is transparent to either party.

Java runtime1.1.8 and above provide SOCKS support. Java.net. Socket instance has the ability to connect to a remote host through the set SOCKS proxy server. If the System property socksProxyHost and optionally socksProxyPort is set, the Socket implementation redirects the connection through the SOCKS proxy Server. Tunneling through proxies, using SOCKS, presents a more generic and viable solution for JMS Applets. As socksProxyPort and socksProxyHost are set as a part of the system property, there are no changes needed in the Client Applet to borrow through the SOCKS server. Single version of an applet can now be downloaded by the Client, irrespective of the Client being behind a firewall or not. The Client Applet automatically fetches the proxy settings from the browser. There are slight variations in the applet and application code that is used to tunnel through the SOCKS Proxy. Using Http tunneling requires the applet to explicitly set the proxy Address and proxy Port in the Applet. The code snippets provided in this document illustrate the proxy tunneling in applications and applets.

This support does not work with JDK versions below 1.4 and 1.5 due to a bug in sun's socket implementation.

Complete samples can be found in the Tunneling Samples folder located in %FMQ_DIR%\fmq\samples\ directory of FioranoMQ installation.

Enabling JMS Applets to Tunnel through SOCKS Proxy Server

Browsers allow users to either manually set the Proxy Server/SOCKS Server Host and port or use a script to automatically set the browser configuration. Applets can leverage support of Java for SOCKS proxy server settings by conveying the settings effectively to the Java VM, used by the browser.

Microsoft Internet Explorer 4.0 and above provide complete SOCKS proxy support and does not require any code changes to run Applets that execute behind the client firewalls.

Netscape Communicator does not convey its proxy server settings to its Java VM. This can be achieved by using digital certificates. A digital certificate allows the Client Applet to set System properties for Java VM of Netscape to use appropriate SOCKS proxy settings. Please refer to the SockPubSub samples directory of your FioranoMQ installation for more information.

Additional Notes on SOCKS

JDK implements SOCKS Version 4. One main problem with SOCKS Version4 is that it accepts remote host address in numeric IP format (and not domain names such as www.fiorano.com). Hence, if the clients are unable to resolve the domain name to its IP address, tunneling does not work. To overcome this limitation, the client must download the Applet from well known IP address, instead of domain names. Another solution is to provide the Server IP Address as one of the Applet parameters.

HTTP Pinging

The HTTP implementation necessitates for the server to detect the clients that have been disconnected from the server. This enables cleaning up the resources for clients that have been disconnected or have timed out. The implementation of this feature requires the client library to continuously ping the server at predefined intervals (configurable through the server configuration). The pinging services are essential for the HTTP support of FioranoMQ and thus are enabled by default in case of HTTP. Due to the basic features of HTTP, there are limitations to detect control-C (or terminating the application abruptly) issues from the client side. It is expected that the JMS client application programmers do an explicit connection.close() for the server to be able to detect that the client is disconnecting so that the resources may be cleaned from the server side. Not doing an explicit close may result in inconsistency in the messages received/delivered.

Chapter 5: FioranoMQ Security

FioranoMQ provides a comprehensive security model that requires minimal application involvement.

The key benefits of FioranoMQ are:

- Complete implementation of Java 2 Security APIs.
- Design and implementation of JMS Applications, independent of the security policy.
- Configuration of security, and user privileges through a central administration tool.

In order to truly leverage the internet as a platform for business applications, organizations need to control their access to corporate assets, such as databases and business rules. Each class of users (such as employees, customers, partners, and suppliers) requires different levels of access.

Information may travel to many locations over the network; yet confidential messages must remain private so that it prevents others from reading and secretly tampering with that content.

This implies a need for two very important facets of security:

Authentication: Determination of user identity

Authorization: Definition and control of user activity

User Identification and Authentication

The FioranoMQ security subsystem supports user identification and authentication using standard JMS APIs. The integrity and privacy of data (discussed in the next section) is protected using MD5 (Message Digest 5) checksums and 40-bit and 128-bit encryption. In addition, FioranoMQ supports destination-based security, allowing setting access permissions for Topics and Queues on the FioranoMQ Server.

FioranoMQ implements the username/password model specified by the JMS API, as described below:

1. Set up various users in the system using the FioranoMQ Administration API/GUI tools. All created usernames are stored in the FioranoMQ off-line database, together with the corresponding passwords and associated descriptions.
2. When a client application tries to connect to the FioranoMQ server using the API: TopicConnectionFactory.createTopicConnection (String username, String passwd)

The FioranoMQ runtime library (embedded in the client) sends a connection request to the server, with the user name and password. The server searches for the user name in its repository. If the user name is found, the server compares the supplied password with the existing password in the repository (set up initially by the administrator). If the password matches, the connection request is accepted, otherwise it is rejected and the client at runtime throws an exception.

If the user name sent in the login packet cannot be found in the repository, the server rejects the connection. A valid connection is allowed if the anonymous user is present in the users list (anonymous user is shipped with the product). In addition, any user can create a connection using the following cases:

- TopicConnectionFactory.createTopicConnection(null,null);
- TopicConnectionFactory.createTopicConnection("anystring",null);
- TopicConnectionFactory.createTopicConnection(null,"anystring");

All these connections are equivalent to the following:

TopicConnectionFactory.createTopicConnection ("anonymous","anonymous") call.

If this option is not wanted, then anonymous user (shipped with the installation) should be deleted through Admin APIs. All such calls do not allow creation of connections. The same is applicable for createQueueConnection() and createConnection() calls.

Data Protection

The secure version of FioranoMQ server protects the integrity and privacy of all the messages exchanged between the client and server.

The integrity feature verifies that the message content on delivery, matches its original published form. Corruption of data can be accidental or intentional. FioranoMQ uses the cryptographic checksum Message Digest 5 (MD5) algorithm to validate the integrity of message content.

FioranoMQ ensures the privacy of a message by using encryption. Encryption scrambles the message content before sending it over the network and restores the original form on delivery. Incase the message is intercepted before delivery (if someone attempts to read it as it travels over the network), then the data is in unreadable form. By default, FioranoMQ encrypts messages to provide privacy using 40-bit encryption, using the Data Encryption Standard (DES) algorithm. However, FioranoMQ allows easy customization of the chosen cipher suite. For example, it is possible to switch from DES 40-bit encryption to 128-bit RSA encryption (domestic version only), or to switch between various available 40-bit encryption algorithms, by setting the SSL parameters.

FioranoMQ 7.0 onwards provides for seamless integration with NT realms. This obviates the need for the Enterprise Administrator to set up separate user realms for MQ. FioranoMQ can seamlessly integrate with existing NT/Solaris realms.

Authentication Based on Digital Certificates

Besides username/password authentication, FioranoMQ also incorporates authentication based on digital certificates. This feature is only available in the secure FioranoMQ server. When certificate based authentication is enabled, each client passes on a one way encrypted version of its digital certificate to the server while trying to establish a connection. The server authenticates the client certificate and if successful, passes back its own certificate to the client process, allowing the client to verify the identity of the server.

Security Realms

FioranoMQ supports a Realm based security that allows FioranoMQ to integrate with Solaris and NT Security realms. This would eliminate the need to create MQ specific users/permissions.

A realm is an administrative entity around which basic operational security policies revolve. A realm determines the scope of the security data and is normally used to organize the objects used in defining access control policies.

Security realms represent a logical grouping of Users, Groups, and ACLs for protecting FioranoMQ server resources. The default security realm or one of the sets of alternative security realms can be used, which allow usage of Windows NT, UNIX, and LDAP [Lightweight Directory Access Protocol] security stores. In addition, FioranoMQ supports custom developed security realms.

A Realm object provides access to users and the main Principals around which a realm is organized, and supports modifying (and extending) it according to policies defined by the realm administrator and by each particular kind of realm. Different Realms use different Authentication Protocols such as passwords (or pass phrases) and public key certificates. Groups of users (and of other groups) are used to define various policies applying to many users. Access Control Lists (ACLs) are uniquely associated with entries in each realm.

FioranoMQ implements a sophisticated security engine that allows dynamic updating of Users/Groups and their privileges. Users, Groups, and ACLs can be retrieved as needed from an external source. FioranoMQ Realms Subsystem is divided in two services: User Management and Access Control Management, each of which is discussed in the following sections.

FioranoMQ User Management

FioranoMQ User Management service uses Realms to retrieve Users and Groups as Java objects.
Any one of the following realms can be chosen for User management:

- Default Realm
- NT Realm
- RDBMS Realm
- LDAP Realm
- Caching Realm
- XML Realm

The User Manager implementation can be specified in the profile deployed during configuration.

Access Control Management

FioranoMQ includes a powerful and flexible access control system to control access to applications and the backend services that clients access through the FioranoMQ Server. The access control system is built on the Java2 security APIs.

An ACL (Access Control List) guards an object or service in the FioranoMQ Server. ACLs can guard Topics and Queues. In addition, custom ACLs can be created for use in applications. An ACL holds a list of ACL entries, each with a set of permissions for a user or group. Permission is actions that can be performed on the protected destination, for example, "publish", "lookup" and "subscribe".

FioranoMQ's dynamic verification engine is invoked before any service call is executed, which checks pertinent ACLs, testing whether the user has the permission required to continue.

By default, FioranoMQ uses the file-based data store for storing ACL information. Access Control Lists (ACLs) are associated with realms in such a way that the entries in them, which identify users and groups, are only significant within a particular realm. FioranoMQ realms are dynamic; they retrieve Users, Groups, and ACLs as needed from an external source.

More information about Access Control List is available in the Java documentation of the java.security.acl package.

Any of the following realms can be chosen for ACL management:

- Default Realm
- RDBMS Realm
- LDAP Realm
- XML Realm

The ACL Manager Implementation can be specified in the profile deployed during configuration.

Default Realm

The default security realm is the File-based Security Storage System in which the User/Group Information and credentials are stored in the file based persistent store.

NT Realm

Using Fiorano NTRealm, defining users and group can be avoided specifically on FioranoMQ. Windows NT Security realm of FioranoMQ uses account information defined for a Windows NT domain, to authenticate Users and Groups. FioranoMQ NTRealm provides authentication using the WindowsNT security domain controller.

Salient Features

Fiorano NT Realm requires that FioranoMQ Server is run as a Windows administrative user, who can read security-related data from the Windows NT Domain Controller. To use Fiorano NT Realm, FioranoMQ must be run on a computer in the Windows NT domain.

To manage user and group information, the FioranoMQ Server must be able to make system calls on the Windows NT computer, where the FioranoMQ Server is running. In other words, FioranoMQ needs appropriate privileges to be able to communicate with the Primary Domain Controller to perform authentication.

In NT Principal Manager, only users registered in Administrators group have rights to open/create AdminConnection. Other users can be given these rights by adding/registering them to default Administrators group.

Note: User admin (used by default to create admin connections) is not a member of Administrators group for FioranoMQ NT Realm. In order to use FioranoMQ default admin tools and APIs, one has to register admin user in the Administrators group

Limitations

A Group cannot have a Group as a member.

Changing password for a user through FioranoNTRealm API is not supported.

It can be done by using the Windows NT Administration Tool.

Note: These are limitations of using NT implementation of realm.principal only and can be over-ridden by using any other implementation of FioranoMQ Realm.

Troubleshooting

The most common configuration problems encountered with Fiorano NT Realm are related with Windows NT policies and specifically the user whose account runs the FioranoMQ Server. The user account that runs FioranoMQ Server requires special permissions to access the Windows NT domain. The steps for granting these permissions are in the configuration instructions.

Another very commonly occurring problem is when FioranoMQ Server is unable to load the file fioranorealm.dll. If FioranoMQ is unable to load the fioranorealm.dll, it gives the following message:

```
java.lang.UnsatisfiedLinkError: no fioranorealm in java.library.path

at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1312)

at java.lang.Runtime.loadLibrary0(Runtime.java:749)

at java.lang.System.loadLibrary(System.java:820)

at fiorano.jms.realm.principal.nt.FioranoNTManager.init(FioranoNTManager.java:82)

at fiorano.jms.realm.principal.nt.FioranoNTManager.(FioranoNTManager.java:51)

at
fiorano.jms.realm.principal.nt.PrincipalManagerImpl.startup(PrincipalManagerImpl.java:
61)

at fiorano.jms.realm.RealmManagerImpl.startup(RealmManagerImpl.java:77)

at fiorano.jms.ex.Executive.startup(Executive.java:647)

at fiorano.jms.ex.Kernel.startup(Kernel.java:61)

at fiorano.jms.ex.fmpmain.main(fmpmain.java:60)

fiorano.jms.common.FioranoException: REALM_NOT_SUPPORTED :: NT realm support is not
available
```

RDBMS Realm

The RDBMS security realm is a custom security realm that stores users, groups and ACLs in a relational database. It uses configuration information to obtain database connection information, from which it connects to a database and loads Users, Groups, Permissions, and ACLs. Since the methods for loading and saving the Realm modify any values, the Realm is not loaded or saved, but is based on stored state in the database.

Configuring the RDBMS Security realm involves setting fields that define the JDBC driver being used to connect to the database. In addition, it defines the schema used to store Users, Groups, and ACLs in the database.

Directory	Description
DbDriver	The full class name of the JDBC driver. This class name must be in the CLASSPATH of FMQ Server.
Url	The URL for the database used with the RDBMS realm, as specified by the JDBC driver documentation
UserName	The user name for the database.
Password	The password for the database user.

LDAP Realm

LDAP Realm provides authentication through a Lightweight Directory Access Protocol (LDAP) server, which allows management of Users, Groups and ACLs in one location, the LDAP directory. LDAP realms allow storage or usage of ACL/user information on any external LDAP server. When the LDAP security realm is used, the LDAP server authenticates Users and Groups.

In the case of SSL protocol (with FioranoMQ Server), the LDAP Security Realm retrieves a common name of the User from its digital certificate and searches the LDAP directory for that name. The LDAP Security Realm does not verify the digital certificate. This verification is performed by the SSL protocol. The LDAP Security Realm currently supports Netscape Directory Server, Microsoft Site Server, OpenLDAP and Novell NDS.

Configuring the LDAP Security Realm

Configuring the LDAP Security realm involves defining the fields that enable LDAP Security realm in FioranoMQ Server to communicate with the LDAP server. In addition, it involves defining the fields that describe how Users and Groups are stored in the LDAP directory. The fields are described in the Table.

Directory	Description
LdapProviderURL	Location of URL server. Change the URL to the name o the computer on which the LDAP server is running and the port number at which it is listening. If it is required for FioranoMQ server to connect to the LDAP server using the SSL protocol, use the LDAP server's SSL port in the URL.
Principal	The distinguished name (DN) of the LDAP user used by FioranoMQ server to connect to the LDAP server. The user must be able to list the LDAP users and group.
Credential	Password that authenticates the LDAP user as defined in the principal field.
LdapsecurityAuthentication	Determines the method for authenticating users.
LdapUserPasswordAttribute	Password of the LDAP user.

LdapUserDN	A list of attributes that when combined with the attributes in the user name attribute field, uniquely identifies an LDAP user.
LdapUserNameAttribute	The login name of the LDAP user. The value of this field can be the common name of an LDAP user, but usually it is an abbreviated string, such as User ID.
LdapGroupDN	The list of attributes that when combined with the group name attribute field uniquely identifies a group in the LDAP directory.
LdapGroupNameAttribute	The name of a group in the LDAP directory. It is usually a common name.
LdapGroupUsernameAttribute	Name of the LDAP attribute that contains a group member in a group entry.

Miscellaneous Features

If caching is enabled, the Caching Realm internally caches Users and Groups to avoid frequent lookups in the LDAP directory. Each object in the Users and Groups cache has a TTL field (TimeToLive), which is set while configuring the Caching realm. If changes are made in the LDAP directory, those changes are not reflected in the LDAP Security realm until the cached object expires or is flushed from the cache. The default TTL is 60 seconds for unsuccessful lookups and 10 seconds for successful lookups. Unless the TTL fields are changed for the User and Group caches, changes in the LDAP directory should be reflected in the LDAP Security realm in 60 seconds.

If some server-side code has performed a lookup in the LDAP Security realm, such as a getUser() call on the LDAP Security realm, the object returned by the realm cannot be released until the code releases it. Therefore, a user authenticated by FioranoMQ Server remains valid as long as the connection persists, even if the user is deleted from the LDAP directory.

Schema checking is turned on by default in the directory server, and Netscape recommends running the directory server with schema checking turned on. The schema checking is turned off for realmLDAP.

XML Realm

FioranoMQ provides XML based Security Storage System in which the User/Group Information, their Credentials and ACL information are stored in XML format.

Caching Realm

The Caching Realms works with the file alternate security or custom security Realms to fulfill client requests with proper authentication and authorization. The Caching realm stores the results of both successful and unsuccessful realm lookups. It manages a separate cache for Users, Groups and authentication requests. The Caching realm improves the performance of FioranoMQ Server by caching lookups and reducing the number of calls into other security realms.

Caching can be used with any supported FioranoMQ security Realm. The separate lists of resources, such as users, groups and users versus passwords are cached. Caching avoids repeated calls to the underlying security store such as NT/ UNIX security store or LDAP.

FioranoMQ Security - Salient Features & Advantages

Design Advantages

FioranoMQ allows developers to focus on building the application and not on implementing a security policy. Security operates independently of application code through an easy-to-use, central administration interface that manages users, groups, Access Control Lists (ACLs). This design permits remote administration for all aspects of security. If security policies of an organization change, the system administrator can manipulate security mechanisms of FioranoMQ without requiring the application developers to rewrite any application code. By allowing security policies to change with business needs, FioranoMQ provides the flexibility that can extend the life of an application.

Effective Protection of JMS Destinations

FioranoMQ achieves this simple yet comprehensive security because it effectively protects JMS Destinations: Topics and Queues. By enforcing security policies on destinations, FioranoMQ allows developers and system designers to indirectly address security through their design of Topics and Queues. Existing applications can immediately take advantage of a new security feature as it becomes available in future versions of FioranoMQ.

Centralized Control

The identification and authentication process is the only area where the client application must address security. The application developer is responsible for the task of soliciting and passing the user name and password to FioranoMQ. This implies that the application must solicit from the user sufficient information so that FioranoMQ can authenticate the user. Beyond this aspect (a standard part of any application), the client application code does not implement or require information for security. Instead, the system administrator uses an external Administration Tool to visually set the security policies, which FioranoMQ enforces.

FioranoMQ and its security subsystem do not require any pre-existing software on the client. All security functionality is built by FioranoMQ and provided through standards-based Java Development Kit (JDK) interfaces. As such, security is built into each and every application or applet that is created with FioranoMQ. Moreover, a developer need not worry about whether the application runs locally or as a downloaded applet. The FioranoMQ security subsystem does not require access to any local (client-side) resources. The security subsystem uses either part of the Java runtime environment or classes downloaded with the applet.

Destination-based Security

In FioranoMQ, all the information flow is based on destinations as illustrated below:

- Developers organize content based on destinations.
- Applications can register their interest in consuming information by subscribing to a destination.
- Applications can produce information by publishing messages to destinations.
- The FioranoMQ Server routes information from publishers to subscribers based on the destination.
- The security subsystem takes advantage of dependency of the information flow on destinations. By protecting the destination, the flow of information can be precisely and dynamically controlled. FioranoMQ refers to this as destination-based security. FioranoMQ associates a security policy with every destination.

Authorization and Access Control

FioranoMQ provides the ability to control the users who can publish, subscribe, or request guaranteed delivery on a particular destination, through the use of Access Control Lists (ACLs). The system administrator uses the Administration Console to define ACLs for specific destinations. FioranoMQ automatically uses these ACLs as described in the following section.

The FioranoMQ Server performs access mediation on publish and subscribe operations of a client and on guaranteed delivery requests. For example, when the client subscribes to a destination, the Server receives the policy for the destination and checks to verify whether the client is permitted to subscribe to the destination. If durable subscription is requested on the destination, an access check is also performed for durable subscription at the time the DurableSubscriber object is created. If either one of these checks fail, the subscribe request is rejected and the client application throws an exception.

When a client publishes a message, on a destination, the server checks to verify, whether the client is authorized to publish on that destination or not . If the client is not authorized, the publish request is rejected and the client application throws a JMSException. If the client is authorized, the server delivers the message to all the clients subscribed to the destination of the message.

Access mediation is completely done on the server side. However, a client can check for permission to publish, subscribe or request guaranteed delivery on a specific destination by retrieving the appropriate ACL object and examining its contents.

The system administrator uses the administration console to add new users, groups, and policies for destinations.

Default Users, Groups and ACLs

By default the following users are created in FioranoMQ:

- Admin
- Anonymous
- Ayrton

Each user is a member of "EVERYONE" group. Depending on the configuration parameter "CreateDefaultAcls" (default value true), the ACLs are created for all the topics and queues on server startup.

Chapter 6: FioranoMQ Data Stores

A messaging system uses a store to save persistent JMS messages. These messages need to be delivered to their intended consumers reliably.

Storage type

Persistent messages can be stored in two ways by using a:

File based store: FioranoMQ has a proprietary mechanism to store/retrieve messages from a flat-file based store.

OR

RDBMS based store: Messages can be stored in a JDBC-compliant RDBMS. Any standard RDBMS like Oracle, MSSQL, MySql, IBM DB2, Cloudscape and HSQL can be used to store the messages.

An administrator can configure the server to store messages in one of these message stores.

By default, FioranoMQ is configured to use a file-based message store. FioranoMQ also provides the support of having different stores for the two messaging domains. There can be one store for messages in the PTP domain and another for messages in the Pub/Sub domain.

The type of store to be used for messages associated with a particular destination can be specified during the creation of the destination. Therefore, there are two ways for administrators to specify the storage type of a destination:

1. By using the FioranoMQ Administrator Console
2. By using FioranoMQ Administration API

File-based store versus JDBC-compliant RDBMS store

The performance of FioranoMQ server with an RDBMS-based message store is lesser than with a file-based message store.

RDBMS based store provides higher security.

Database reliability is generally higher.

Database stores may generate network traffic if the database server is on a different JVM or machine. This does not arise in the case of a file-based store.

Default Destinations for sample applications

FioranoMQ creates default destination objects that can be used by client applications to run the samples provided with the installer. FioranoMQ is configured to use the file-based message store by default.

In this configuration, only the file-based default destination is created.

It is possible to configure an instance of FioranoMQ Server to use the file-based message store and the RDBMS-based message store.

In this case, file-based and RDBMS-based default destination objects are created.

File-based default destination objects are:

- PrimaryQueue
- SecondaryQueue
- PrimaryTopic
- SecondaryTopic

The messages published on these destinations are stored in a file-based message store.

RDBMS-based default destination objects are:

- PrimaryRDBMSQueue
- SecondaryRDBMSQueue
- PrimaryRDBMSTopic
- SecondaryRDBMSTopic

The messages published on these destinations are stored in the RDBMS-based message store.

By enabling both file-based and RDBMS-based stores in a single instance of FioranoMQ server, messages that require fast retrieval can be saved in a file-based store whereas messages that require the reliability of JDBC-compliant relational databases can be stored in an RDBMS-based file store.

Creating a default database

FioranoMQ can be configured to store the messages in a JDBC-compliant relational database. By default, FioranoMQ is configured to use the HSQL database. Fiorano ships the library containing the JDBC driver for HSQL with the FioranoMQ product. Incase a different RDBMS is required; administrators need to create the FioranoMQ database in their RDBMS. If FioranoMQ needs to store messages in an Oracle database, administrators are expected to create database tables for storing the messages and related information in the Oracle database. FioranoMQ provides scripts that should be used to create the required tables. These scripts need to be executed before running the RDBMS enabled FioranoMQ server. These scripts accept various arguments such as URL, UserName and Password as system variables and then create the file-based or RDBMS-based databases.

A FioranoMQ database can be created using one of the following two approaches:

- Command Line parameters
- Using Fiorano Studio

Clearing a database

A script is provided for clearing both file-based and RDBMS-based databases.

Chapter 7: Managing Administrated Objects

JMS Entities like Destinations & Connection Factories (also collectively known as administered objects) are well defined from a usage point of view in JMS Specifications. The roles of these objects as well as the APIs that can be used to operate on them are well defined. However JMS specifications do not talk much on how to obtain an instance of these objects. This decision is therefore left up to the JMS Provider on how it gives an instance of these objects to an application.

FioranoMQ allows applications to use JNDI APIs to obtain instances of Administered objects

Note: For additional information on JNDI, please refer
http://java.sun.com/products/jndi/

This approach allows application code to be "Standards Based" and is therefore not necessary to import any Fiorano specific classes. Further for user convenience, FioranoMQ provides a limited implementation of the directory server. This is done so that an end user is not required to configure a third party naming & directory service for using FioranoMQ.

FioranoMQ also allows applications to store and retrieve administered objects from any third party medium example, an external LDAP server. Besides this, FioranoMQ allows its applications to create a new instance of an administered object (destination or queue) and use the same in all JMS operations. This requires, using non-standard Fiorano specific APIs in the application.

Naming Services

The Naming Manager, a module within the FioranoMQ server, provides all common Naming Services (lookup, bind, delete, list etc). Naming requests, originating from a JMS Application, that are sent to FioranoMQ Server are internally forwarded to this module, which processes the request and responds accordingly. Administrative requests leading to creation or deletion of administered objects are also forwarded to this module.

The interface for this module is well defined and this allows having multiple implementations of the same. Various implementations (as summarized below) differ on the persistent media used for storing information. FioranoMQ Administrator is free to plug in any implementation (or even write a new implementation and plug it in) of this interface in the server.

File

This is the default implementation for Naming Manager. It stores information in a proprietary File (defaults to "admin.dat" in run folder of the profile).

XML

This implementation stores information in clear text format in an XML file (defaults to "admin.xml" in run folder of the profile).

LDAP

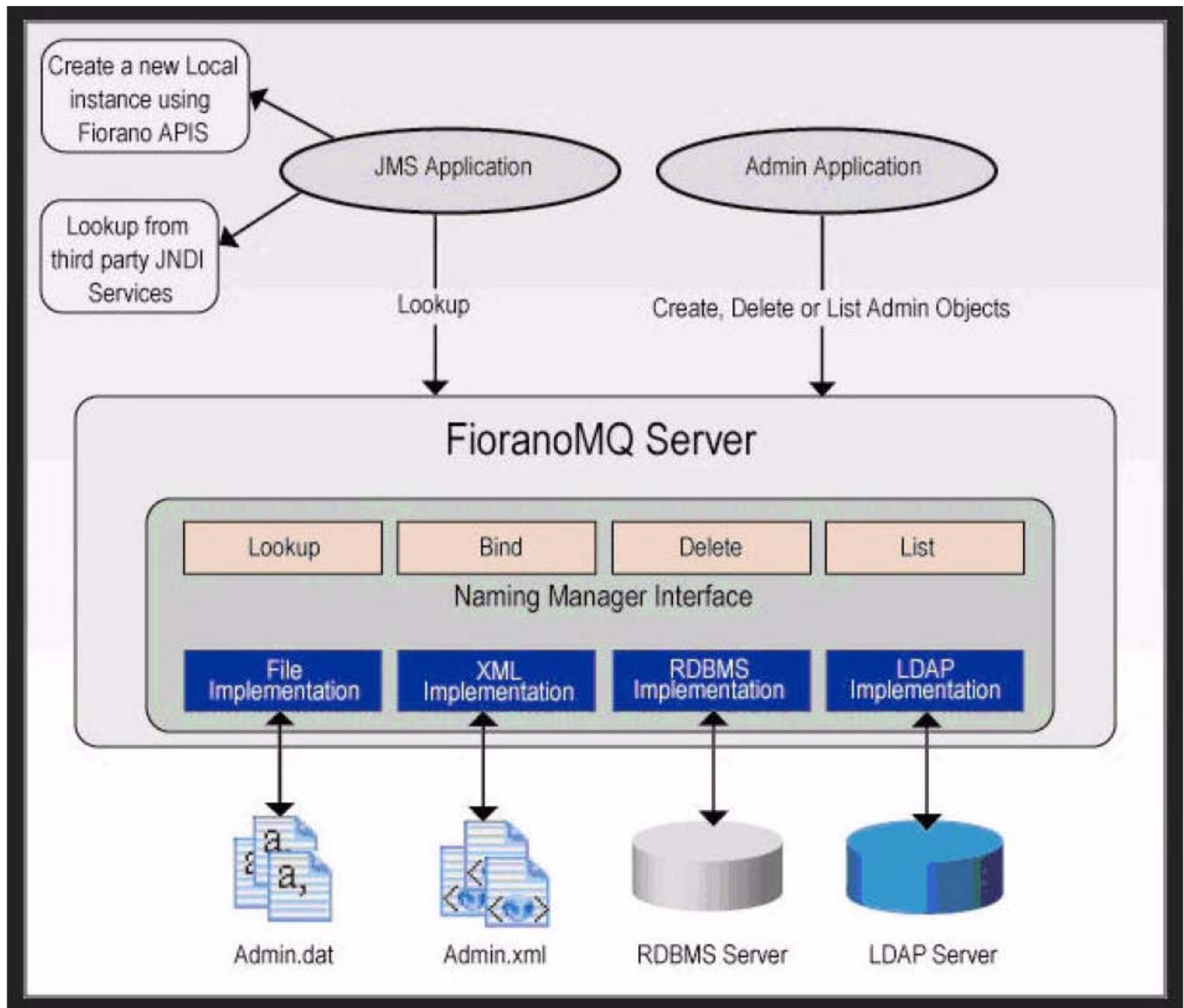
This implementation uses any third party JNDI compliant Naming & Directory Service to persist information.

RDBMS

This implementation uses any third party JDBC compliant RDBMS server to persist information.

Cache

This implementation creates a "cache" of admin objects in-memory. It can use any of the above-mentioned implementations underneath for storage purposes. This implementation is beneficial to use in situations where there wouldn't be many changes frequently in the admin object store.



Salient Features

FioranoMQ can be configured to use any of the above-mentioned Naming Manager implementation. This can be done only through off-line configuration.

The JNDI implementation provided by FioranoMQ is limited and provides implementation of only basic methods. It should not be considered or used as a full- blown JNDI implementation.

Chapter 8: Message Expiry

JMS Standard allows setting a Time To Live (TTL) on a message at the time of sending it to a destination. The JMS provider is required to consider this message as valid only till the specified TTL. Once the TTL time elapses, the message is considered to be expired, and would not be available on the destination and hence wouldn't be delivered to any consumer.

Point of Checking of Message Expiry

The server adds the TTL specified in a message to the current time to obtain the expiry time. This is done at the time when the message fist enters the server. The server then checks for expiry when attempting to deliver it to a consumer. If expired, the message is ignored.

Since expiry is checked just before delivering the message to a consumer, in case there is no active consumer, expired messages might continue to consume server's resources (disk or memory space). In order to avoid this, the server can be configured to check expired messages in all queues periodically. This can be done by setting the value of flag DbCleanupEnabled to true (by default it is set to false). The frequency with which the server checks for expired messages is again configurable through the parameter CleanupInterval (defaults to 10 minutes).

On Detection of an Expired Message

Once the server detects an expired message, it deletes this message from the destination as it is no longer useful for any consumer. Since this deletion is done automatically in the server, the following additional actions at this stage can be performed to inform an interested application of this event.

Copy of a message pushed into Dead Message Queue

Copy of the message is published on Admin Topic

Note: Publishing on admin topic has been deprecated and would be discontinued in future releases. The server fires a JMX Notification with information about the expired message.

The sections below provide more detail about these actions.

Dead Message Queue

Dead Message Queue is a special system queue with the name SYSTEM_DEADMESSAGES_QUEUE created for storing copies of messages that expire in any of the server destinations. Any client applications can then browse or receive messages from this queue using normal JMS semantics.

DMQ Configuration

Controls are provided to configure DMQ functionality globally for all queues as well as for an individual queue. These controls are:

Parameter	Scope	Possible values
EnableDMQOnAllQueues	Global	Yes, No
EnableDMQ	Individual Queue	Yes, No, Default

The following flow-chart describes the steps involved in checking if DMQ functionality is turned on or not for a particular queue. By default individual queues have enableDMQ set to "Default". This allows the administrator to control DMQ configuration for all queues through global flags.

Other configuration parameters with respect to DMQ are summarized in the table below:

Parameter	Description
DMQExpiryTime	The time for which messages would live on DMQ.
CleanupDmqAtStartup	If set to Yes, all DMQ messages would be deleted on server startup.

Selectively disabling DMQ for a message

If DMQ is enabled for a destination, all messages that expire are by default added to DMQ. However if an application doesn't want to use DMQ functionality for specific messages, it can do so by setting properties in message through various APIs.

Message Expired Notifications

When a message has expired, the server (if configured) publishes a notification in the form of a JMS Text message on a system topic named ADMINISTRATOR_TOPIC. This functionality can be used to get notifications whenever a message expires. Any application can create a subscriber as per JMS semantics to receive these notifications.

Configuration

Notifications can be configured globally through a flag EnableNotificationOnDeadMessage. If, it is set to be true, the server publishes a notification whenever a message expires. However if an application wants to disable this functionality for specific messages it can do so by setting properties in message through APIs.

Salient Features

Notifications work only if DMQ is configured for the queue. The published Text Message has the following attributes.

It has the same set of properties as the original message that expired.

Its body contains (as text) the destination name on which the original message was pushed. It is pushed as a non-persistent message. This support is deprecated and future releases fires JMX Notifications when a message expires.

Chapter 9: Snooper

Snooper is a FioranoMQ feature that allows an application or an individual to view incoming messages arriving on a destination. The intended use of this feature is to facilitate debugging of JMS applications.

Snooper functionality can be used through Fiorano Studio (Customized GUI Tool). This tool allows the administrator to enable/disable snooping functionality on destinations and can also show the contents of a "snooped" message in a tabular manner in the GUI. A console-based application can also be written in order to snoop messages. This application can programmatically receive the published message and inspect the same in any manner.

Snooper Configuration

In order to snoop messages on a destination, snooping functionality has to be turned on for that destination. This can be easily done through Studio as well as programmatically through a Java Application that uses Admin APIs.

Besides enabling/disabling this functionality for a destination, one can leave Snooper configuration on a destination to be "Default". With this configuration, global flags decide if snooping is be turned on or off for that destination. This provides the flexibility of turning on or off snooping functionality on all queues or topics at the same time. To summarize the checks for snooper functionality is checked via the following flow chart.

Note: The values of global parameters are consulted only when snooper configuration on a destination is set to be "Default". For other values (on/off) the global parameters are ignored. All FioranoMQ destinations on creation are configured to this value.

Working of Snooper

If snooping were turned on, FioranoMQ Server would simply send a copy of the incoming message to pre-configured system topics. An application can then pick up this message and inspect the same.

The System topics used for this purpose are:

- SYSTEM_MESSAGE_SNOOPER_TOPIC
- SYSTEM_MESSAGE_SNOOPER_QUEUE

A message coming on a topic is sent on the first, while the latter is used for messages coming on a queue.

Security Settings

Security Settings for Snooping are controlled by the ACLs of the above mentioned system topics. By default the following restrictions apply.

Durable subscriptions are not allowed.

FioranoMQ administrator alone can "snoop".

FioranoMQ Administrators can edit the ACLs of these topics programmatically to modify the above restrictions. The ACL name to use would be the same as the names of the system topics described above.

Miscellaneous Features

The following are important points that a user should remember.

"Snooped" Messages are a copy of the original messages arriving on a destination. Making a change in this message would not affect the actual message.

"Snooped" Messages always have their delivery mode set to "Non Persistent" even if the original incoming message was a persistent one.

Snooping is not permitted on system topics.

Chapter 10: Durable Connections

Network reliability is a common problem faced when designing a system that is spread over multiple machines. Enterprises across the world spend a large amount of time and resources on network management, but the fact still remains that network links cannot be 100% reliable. Mission critical applications cannot afford to lose data in any eventuality and must always be built with this premise. This requires the application programmers to build a store and forward layer in their Application infrastructures. This store and forward layer involves storing precious data upon detecting a network loss, take corrective action and then send the previously cached data again.

JMS standard requires middleware to build store and forward mechanism for consumers. This is achieved by marking a consumer as durable. If a consumer is unavailable due to any reason, the server is required to hold onto its messages. These messages are delivered to the consumer once it becomes available. This standard JMS feature ensures that a durable consumer always receives its messages. However, JMS does not provide similar reliability for a producer. This implies that if the server is unavailable due to any reason (such as network problem and high loads) then the send mechanism of the producer fails, resulting in an appropriate exception. This forces applications to implement a store at their end and transfer this data when connectivity is restored.

FioranoMQ provides enhancements over the JMS support to provide the store and forward functionality. This functionality allows JMS applications to continue all their operations even if the server is un-available due to any reason. This frees up the application from all the network related troubles, as it no longer has to take care of them. A network disruption is not visible to a JMS application built over FioranoMQ.

Overview

FioranoMQ introduces the concept of a Durable Connection. A Durable Connection maintains connectivity with the FioranoMQ server at all times. The applications do not have to take care of storing, re-connecting and then forwarding the stored messages to the server. This frees up the application from the complex task of building the store and forward mechanism at its end.

In addition, it improves the reliability of the underlying JMS transport. If due to any reason the connection is lost and the application fails to transfer data, Durable Connections try to restore the connection automatically. In addition, they ensure that data is not lost in transit and is sent as soon as the connection is re-established. These activities are not visible to the application and are performed automatically by Fiorano's runtime library upon detecting the failure. This makes the system highly reliable and robust even in the case of network failures.

For example, consider a process computer monitoring a steel mill. Real time steel production information is sent each second to a main hub. The main hub uses this information to generate the desired results. If the connection between the Process machine and the Hub breaks, the send mechanism fails and an exception would be raised. Since this data is generated only once, the application is required to store this data at its end upon encountering the exception and then spend resources to connect back to the server. This adds considerable load to the application.

In such cases, a Durable Connection comes to the rescue as it does all the hard work for the application. It automatically tries to re-establish the connections, stores the data in transit and sends it to the server as soon as the connection is restored.

Note: Durable Connections is a proprietary feature of FioranoMQ while durable subscriptions are a part of JMS specification.

Working of Durable Connection

A durable connection works like an ordinary connection as long as the connectivity is maintained with the FioranoMQ Server. In case, the underlying socket is broken due to any reason, a durable connection performs the following activities:

- Initiates a thread that continuously tries to re-connect with the server
- Initializes a store, on the local machine, to store any new messages that are published.

Both these activities are not visible to the client application and are automatically performed by FioranoMQ's runtime library. When the connection is restored, messages stored in the local store are automatically sent to the server.

Producer on a Durable Connection

A message producer created over a durable connection can send messages irrespective of the fact whether or not it is connected to the server. A producer is able to send messages even if the underlying connectivity does not exist. The runtime library automatically handles any problem in the connectivity.In the eventuality of break down of the underlying connection; the runtime library establishes a local cache of messages on the client's machine. This local cache is used to store messages that are published by the producer during the period it was disconnected from the server. The base directory of this local cache can be configured by the client application. In the base directory, a subdirectory for each connection using this cache is created, where messages for that particular connection are stored. The client application can use any number of durable connections over the same base directory. Once the connectivity has been re-established by Fiorano's runtime library, it transfers the messages stored in the local cache to the server. While transferring the messages, the ordering is maintained for messages sent over a connection.

An application is free to send messages to more than one JMS Destinations over a single Durable Connection. The producers can be created on a transacted as well as a non-transacted session.

Note: Messages are stored in the local cache, irrespective of the DeliveryMode, which implies that both Persistent and Non-persistent messages are stored in the client side cache.

Consumer on a Durable Connection

Since consumers themselves can be defined as "durable" by virtue of their definition in JMS standard, very little is required to be done to ensure that all the messages are delivered to a consumer, even if it was temporarily unavailable. In case, a consumer is created over a Durable Connection, it does not hold the responsibility of reconnecting to the server on losing its connection. In case the underlying connection with FioranoMQ server breaks due to any reason, Fiorano's runtime library spawns a thread that tries to re-connect with the server. Message delivery is restored when the connection is established again.

Advantages

Durable Connections support in FioranoMQ provides a host of advantages over standard JMS implementations. They are as follows:

Network reliability

Durable connections provide network reliability by storing messages on the client side when the server is down. In real world scenarios this is very essential for any application.

Store and Forward capabilities

Durable Connections demonstrate store and forward capabilities even at the client level

Transparent reconnection code

If Durable Connections is enabled, the client application does not hold the responsibility of reconnection logic. It is handled internally by FioranoMQ's runtime library.

Message browsing of persisted messages

FioranoMQ provides a Message Browser through which messages that have been stored in the data store of the client, due to network failure, can be browsed.

No vendor lock in

The revalidation logic is transparent to the client application and is handled by Fiorano's runtime library. Reconnection code and other details do not have to be taken care of. Only the connectionfactory with AllowDurableConnection set needs to be looked up. The client side persistence and reconnection code is handled transparently by Fiorano's runtime library.

Enabling Durable Connections Support

The ability to create a durable connection with the server can be controlled both at the server and the application level. Durable connections can be enabled/disabled using Fiorano Studio by the following two methods:

1. Creating new connection to the server
2. Adding a new connection factory

Disabling Durable Connections in the server configuration disables it universally, for all the clients whereas disabling Durable Connections in a connection factory disables it for all the clients using this connection factory.

Client side Message Cache

A Durable connection creates a cache on the local machine, in case a producer created on it is unable to send a message due to unavailability of the server. The base directory of this cache is configurable as explained in the preceding section. Within this base directory, a subdirectory is created for each connection, using this base directory. The subdirectory is created, with its name same as the Client ID of the durable connection.

For example, if the base directory for durable connections is specified as c:\\temp\\db in myConnectionFactory, any connection created through myConnectionFactory creates its cache in c:\\temp\\db. If there are two connections on the same machine, with clientIDs as "client1" and "client2", the directory structure are as follows:

```
c:\\temp\\db

   |_____ client1.ptp
```

|_____ client2.ptp

Note: In case the Client ID is not set, at runtime FioranoMQ internally creates a unique ID for that connection and a directory by this name is created for client side caching.

However, it is recommended to use Client ID because of following reasons:

If the application wants to send any pending messages when it restarts, it is unable to do so as a new ID is generated for that connection. Since the ID is a complex string, it is difficult to use CSP Message Browser to browse for client side persisted messages.

The messages of a client are identified by its clientID. In case, the client application is terminated due to any reason, the next time this application is started up, the runtime library checks if there are any pending messages stored in the local cache for the connection (this check is performed on the basis of the client ID set on the connection). On finding pending messages, it sends these messages to the server. This operation is performed when the clientID of the connection is set by the application. In case the application wishes to ignore previously cached messages, it needs to add the following flag in the Hash table passed as environment to InitialContext used for looking up operations.

"DONT_SEND_PREVIOUSLY_STORED_MESSAGES", "TRUE"

Use the following API available in the connection, if the client applications need to exercise control over the instant when the pending messages are sent:

```
public void sendPendingMessages ()
     throws JMSException;
```

When the preceding method is invoked on a connection, the runtime invocation sends all the pending messages for this connection to the server.

```
public void purgePendingMessages ()
      throws JMSException;
```

The preceding method is used to purge all the messages in the local cache, which were published on the associated connection.

Note: Both the APIs require casting of JMS Connection into fiorano.jms.runtime.ptp.FioranoQueueConnection or fiorano.jms.runtime.pubsub.FioranoTopicConnection.

Serverless Environment

In few situations, it might be necessary to run the client application in a server less environment. This implies that the client needs to connect to a server even if the server is not available. This might be essential in situations where there is a high probability of server being down when a client tries to connect to it and it is essential that the connection be established.

For example, consider the case of a cellular services provider. The services provider has an SMS gateway, which interacts with the mobile devices by acting as an interface between the JMS server and the mobile phone. The JMS server routes the data ahead. A user uses his mobile phone/PDA or any other hand held device to send an SMS to another user. This message would first reach the gateway, which has the responsibility of routing this message to another gateway interacting with the mobile device of the recipient, through a JMS server. There is a possibility that at the time the message was to be forwarded to the JMS server, the server was down. If such a situation arises, in a normal case that SMS would get lost.

Alternatively, if the gateway receiving the message from the sender is considered to be a JMS client with Durable Connections enabled, then it stores messages locally when the server is down. This would provide a robust and reliable solution in scenarios where the gateway sends messages even when the server is down. These messages would be stored in the local cache and subsequently be routed through the server, when it becomes available again.

To enable a client application to run in a server less environment, set the following in the application:

```
env.put(FioranoJNDIContext.AllowDurableConnections, "true")
FioranoJNDIContext ic = new FioranoJNDIContext(env);
QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookupQCF("primaryQCF");
QueueConnection qc = qcf.createQueueConnection();
qc.setClientID("myClient");
QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWL-EDGE);
Queue queue = qs.createQueue("primaryQueue");
```

This would enable the client application to run with durable connections enabled in a server less environment.

Note: In this case the lookup of the ConnectionFactory is performed by the lookupQCF() method and the Queue is created using createQueue method of the session. In this case the lookup of the ConnectionFactory is performed by the method lookupQCF(). When the server is up, the connection gets revalidated and if the connectionfactory exists on the server, the actual lookup is performed through the server and messages are sent to the appropriate destination from the local cache. In case, after revalidation it is found that the server does not allow durable connections, then the pending messages are not sent to the server.

Sample Application

Sample applications are available in the following directories in the FioranoMQ installation.

%FMQ Home%\fmq\samples\ptp\Durable Connections

%FMQ Home%\fmq\samples\pubsub\Durable Connections

Alternatively, these samples can be downloaded from www.fiorano.com

Relationship with Revalidate

In this model of Durable Connections, the client application does not hold the responsibility of reconnection logic. If Durable Connections is enabled, the entire process is handled internally by FioranoMQ's runtime invocation. If the client application has Durable Connections enabled, then in case the server breaks down, the client does not have to take care about revalidation code. The FioranoMQ runtime invocation internally detects network failure and starts a reconnection thread, which reconnects the client to the server when it is available. This process is not visible to the client application and reconnection is handled internally.

Relationship with CSP

Client Side Persistence was provided in the 6.0 release. These needed proprietary APIs of FioranoMQ. In this release, these APIs have been deprecated.

To enable durable connections, one no longer needs to enable client side persistence. Earlier, in CSP, following had to be set in the client application to enable client side persistence:

```
env.put(FioranoJNDIContext.ENABLE_CLIENT_SIDE_PERSISTENCE,"true")

env.put(FioranoJNDIContext.CSP_BASE_DIR,"c://myCache");

InitialContext ic = new InitialContext(env);

QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("primaryQCF");

QueueConnection qc = qcf.createQueueConnection();

(FioranoQueueConnection)qc.setCSPConnectionID("myClientID");
```

However, in Durable Connections, all these steps do not need to be performed. The client application can lookup a ConnectionFactory, which holds information about the Durable Connection being enabled; and set a unique clientID for the connection on which Durable Connections is to be enabled. The messages sent by the sender would now be stored in the directory structure specified in the ConnectionFactory (or in the ".\CSPCache" directory, if not specified) and sent to the server once the connection is revalidated. All this is handled internally by Fiorano's runtime library.

To enable durable connections, the client application has to set to allow durable Connection in the env as a property and set a client ID for the connection as follows:

```
env.put(FioranoJNDIContext.AllowDurableConnections, "true");

InitialContext ic = new InitialContext(env);

QueueConnectionFactory qcf = (QueueConnectionFactory)ic.lookup("primaryQCF");

QueueConnection qc = qcf.createQueueConnection();

qc.setClientID("myClientID");
```

The messages would get stored on the client machine either in the directory specified in the env variable or in ".\CSPCache". Now there is no need of calling an explicit setCSPConnectionID on the connection on which CSP has to be enabled, but only set a ClientID for the connection to enable Durable Connections.

Constraints in Durable Connections

If more than one application needs to simultaneously use the same local cache, then, please ensure that unique Client IDs are used.

Using Browser for Client-side persistence when an application is using the same local cache can result in aberration in behavior of the browser. It is recommended not to use the Client-side persistence browser when an application is using the same local cache.

Messages can get redelivered with appropriate JMSREDELIVERED flag set. For instance, when the data reaches the MQServer and the client loses connection before the server acknowledges the receipt of data, then messages can get redelivered.

Base durable connection directory cannot be added when creating the connection factory through the Admin GUI.

Chapter 11: Hierarchical Topics

Developers need to organize their data based on its content. JMS accomplishes this by funneling messages to various destinations. However, these destinations do not have any logical correlation with each other.

FioranoMQ provides a way of correlating various destinations. FioranoMQ Destinations can have a parent child relationship. A Topic can be created within a Topic. This results in a hierarchical tree, where each leaf represents a unique topic.

Need For Hierarchical Name Spaces

Topic name spaces offer the ability to organize various destinations in a hierarchical manner. An enterprise may choose to define various levels of hierarchy, depending on the organization of data that would flow on these destinations. Thus, a hierarchy of topics is easier for the client to manage. Moreover, since there is a well-defined relationship in such a hierarchy, it results in an efficient handling of destinations by the provider as well. While a topic hierarchy can be flat (linear), it would typically build from one or more root topics, adding other topics in levels of parent-child relationships to create a hierarchical naming structure.

Name Space Notation

Hierarchical name spaces of FioranoMQ use the same notation as fully qualified packages and classes. This implies that various levels in the hierarchy are distinguished by period-delimited strings. For example, a topic name fiorano.sales.fmq would mean the hierarchy as shown in Figure below.

The hierarchy gets defined automatically at the time of creation of various topics. The process of creating a topic would be successful only when the parent topic exists. This ensures that nodes are added to the hierarchy tree in an orderly manner. FioranoMQ allows the hierarchy to have unlimited number of levels and unlimited number of nodes in a particular level.

Creating Hierarchical Topics

Hierarchical topics can be created like any other topic. A topic at a particular level can be created only if its parent exists in the hierarchy. The first node of a hierarchical topic is called the root node of the hierarchy. Some important features with respect to hierarchical topic names are mentioned below

Case Insensitive

Topic names are case insensitive For example; FioranoMQ considers "ACCOUNTS" and "Accounts" as the same topic.

Spaces in Names

Topic names can include the space character. A space is considered as just another character in the name of a topic. For example, "company.fiorano.comment.FioranoMQ is fast" is a valid topic name. However, the topic names are trimmed before creation.

Empty String

No level in the topic hierarchy can be an empty string. In other words a topic name cannot have two simultaneous dots, for example, "company.fiorano.dept" is an invalid topic name.

Unlimited Length of Topic Names

FioranoMQ supports infinite length of a topic name. This implies that a node in the topic hierarchy can have any number of characters.

Unlimited Depth of Topic Hierarchy

FioranoMQ supports indefinite depth of the hierarchy tree for any topic. As many nodes within a topic can be created.

Wild Card Support

Wildcard character, asterisk (*) or (#) cannot be used in topic names during the creation of hierarchical topics.

Dynamic Creation of Topics in Hierarchy

If a topic is created on running server instance and it's name matches with any subscription expression (if exist) then this topic would become the member of the maintained hierarchy for subscription on Hierarchical topics.

Example: Subscription expression: ABC.*

Topics existing on system: ABC, ABC.1, ABC.2, ABC.1.1

A subscriber looked up topics with expression ABC.* and it is receiving the messages from the matched topics. At runtime a new topic named as ABC.3 is created then this topic would become the part of the hierarchy and published messages on ABC.3 would also received by Subscriber created on ABC.*.

Note: For this feature, events should be enabled at server side.

Looking up Hierarchical Topics

Client applications can lookup a topic in the FioranoMQ server using either JNDI APIs or a bound object of type FioranoInitialContext.

Criteria for looking up hierarchical topics are as follows:

The topic being looked up contain a wild card character either * or # with any number of delimiters (.). The lookup call succeeds only if the root topic has been created earlier by the administrator. If the topic being looked up contains a '*' or '#' then this call would be successful only when there is at least one topic exist in the server whose name matches the criterion.

For example: If user tries to lookup "primarytopic.a.*" or "primarytopic.a.#" then the lookup is successful only if "primarytopic.a" exists.

Publishing on Node(s) in Topic Hierarchy

A Publisher can publish only on a fully specified topic. Publishing on a topic that contains an asterisk (*) or (#) throws an exception.

Subscribing to Node(s) in Topic Hierarchy

Subscriptions are created in the JMS defined manner using the TopicSession. The normal createSubscriber APIs, provided by JMS, can be used to create subscriptions on hierarchical topics.

A subscriber can subscribe to multiple topics using wildcard character.Subscribing to a topic contains a valid wild card character effectively creates a subscribers on all the nodes matched against the expression in hierarchy and uses these subscribers for subscription on Hierarchical topics.

Template Characters Used in Subscription

Wild card characters are the special characters used in creation of the hierarchy of topics. In the topics hierarchy, these characters are referred to as Template Characters. The period (.) delimiter is used together with the asterisk (*) and the pound (#) template characters when subscriptions are fulfilled. Using these characters avoid having to subscribe to multiple topics in server. Client applications can use template characters when subscribing to a set of topics or binding a set of topics.

There are two FioranoMQ template characters used in Hierarchy:

Asterisk (*) FioranoMQ uses two types of conventions for this template character:

An asterisk (*) is the last template character in subscription expression Subscription is made for the root node and all its subordinate nodes in hierarchy.

For Example: If the used expression for subscription is ABC.*, then ABC and all its subordinate topics would be used.

An asterisk (*) is the intermediate character in subscription expression In this pattern root topic is not selected and meaning of * is considered as "one or more occurrence of character ". Example: If the used topic name for subscription is ABC.*.1 then ABC would not select and all the topics, whose name matches with this pattern, get selected and used for subscription as: ABC.1.1, ABC.1.1.1, and ABC.2.1.

Pound (#) select all the topics at one level down in hierarchy. If '#" character is present in the pattern, then all the topics at one down level in hierarchy are used for subscription. Example: If the subscription topic name is ABC.# then all the topics one level down of ABC in hierarchy would be used. Example: ABC.1, ABC.2..Topic Name as ABC or ABC.1.1 doesn't match this pattern.

The intent of the template characters is to allow a set of managed topics to exist in a message server in a way that lets subscriber choose broad subscription parameters that include preferred topics and avoid irrelevant topics.

Constraints using in template characters are as follows

At any node level, a template character precludes using other template characters. Example: Qualify the selection against a pattern such as: A.B*.1 is not allowed. It should be A.B.*.1. In the used pattern, each character should be separated with delimiter (.).

Template characters used for replacement are not allowed (like " ? ").

Except "*", "#", "." no other wild card character is used for subscription to multiple topics.

Conventions used in Hierarchical topics

In Fiorano hierarchical topics only two template characters are used. These characters are # and * with delimiter character (.).

Usage for these template characters is as follows:

Usage of asterisk (*)

Subscription Expression ABC.*

Convention used If * is the last character in subscription expression and no other template character is used with * then root topic and all other topics (where * is replaced with one or more occurrences of any character) would be selected for subscription on Hierarchical topics.

Example

Topics exist in MQ Server: ABC, ABC.1, ABC.2, ABC.1.1, and ABC.1.2

Used expression for subscription: ABC.*

Matched topics: ABC, ABC.1, ABC.2, ABC.1.1, ABC.1.2

Subscription topic name ABC.*.1

Convention used If * is the intermediate character in expression then all other topics (where * is replaced with one or more occurrences of any character in name) would be selected for subscription. In this selection root topic doesn't include.

Example

Topics exist in MQ Server: ABC, ABC.1, ABC.2, ABC.1.1, and ABC.1.1.1

Used expression for subscription: ABC.*.1

Matched topics: ABC.1.1, ABC.1.1.1

Usage of pound (#)

Subscription topic name ABC. #

Convention used If # is the only wild-character present in expression then all topics (where # is replaced with only one occurrence of any character in name) would be used for subscription.

Example

Topic exist in MQ Server: ABC, ABC.1, ABC.2, ABC.1.1, ABC.1.1.1

Used expression for subscription: ABC. #

Matched topics: ABC.1, ABC.2

Subscription topic name ABC#.1

Matched topics ABC.1.1, ABC.2.1, ABC.3.1

Usage of combination of template characters In subscription expression both template characters are also used.

Subsciption topicName ABC.*.#

Convention used such topicname is invalid. As there is no meaning of # character after * character in an expression.

Subscription topicName ABC.#.*

Convention Used In this expression all the topics (where # is replaced with one occurrence of character and * is replaced with one or more occurrence of character) would be used for subscription.

Example

Topic exist in MQ Server: ABC, ABC.1, ABC.1.1, ABC.2, ABC.2.1

Matched topics: ABC.1.1, ABC.2.1

Deleting a Hierarchical Topic

Deletion of a topic/subtopic from the hierarchical name space depends on the value of the parameter AllowDeletionOfSubTopics, which can be configured through Fiorano Studio. If this value is set to true, then deletion of a topic/subtopic deletes all the children of this topic/subtopic. However, if it is set to false, an exception is raised indicating that the user first needs to delete the children of the topic\subtopic before deleting it. By default, this variable is set to false.

Publish/Subscribe across Servers

FioranoMQ supports hierarchical topics across servers. The hierarchical topics across servers can be used in exactly the same way as they are used on a single server.

Security Considerations on Hierarchical Topics

FioranoMQ supports ACL settings for Hierarchical topics. An ACL can be set for any topic, irrespective of the level at which this topic exists. These ACLs are checked at the time of creation of a publisher as well as a subscriber. While creating a subscriber on multiple topics (a topic that involves a template character in its name), the ACLs for all the subtopics are also checked. In addition, the subscriber is modified so that it does not receive messages from all the subtopics that have a negative permission set for the particular user.

Limitations

Topic names cannot contain a wildcard character in topic creation. For subscription expression no other template character (* or # with any number of dots ".") is used. Usage of any other template character throws exception in looked up.

Deletion of a hierarchical topic that has active publishers/subscribers deletes the hierarchical topic. Care needs to be taken not to delete the hierarchical topic, while it contains any active publishers or subscribers.

A publisher cannot publish on multiple topics. A publisher has to specify the complete name of the hierarchical topic on which it wants to publish data. Creation of a publisher on a topic, which contains an asterisk, should throw an exception. Similarly, an exception should be thrown if a publisher tries to publish on a topic, which contains an asterisk.

If a subscriber subscribes on hierarchical topics with subscription expression and while receiving the messages, if the administrator changes the ACL of one of the children of hierarchy then the subscriber would not affected by this change. Creating the new subscribers with subscription expression would be affected by this change.

There is a performance degradation associated with hierarchical topics. So users are advised not to use hierarchical topics for applications where performance is a major requirement.

Chapter 12: Message Encryption

Message encryption allows transfer of sensitive data from one point to another in a secure way. Encryption implies the transformation of plain text into cipher text that is not possible to read without the use of a "key". A key is also used to decrypt the cipher text into plain text.

Base Implementation

FioranoMQ 7.1 and upward support encryption. DES (Data Encryption Standards) is used as the default encryption algorithm. FioranoMQ intends to support more encryption algorithms in its future releases.

There are two types of encryption algorithms:

- secret key algorithms
- public key algorithms

In secret key algorithms, both the sender and the receiver need the same key for encryption and decryption respectively. In public key algorithms, one key, the public key is used for encryption and is published and another key, the private key, is used for decryption. No secret information is exchanged. The private key is mathematically related to the public key. Theoretically it is therefore possible to compute the private key based on the public key. But this is overcome by making the computation as complex as possible. DES is based on secret key cryptography.

Some of the advantages of secret key cryptography as compared to public key cryptography are computation speed is faster. Therefore, this method is recommended for bulk encryption and is commonly used. The encrypted text is compact.

The disadvantage is that administration of keys can become complicated because of sharing of the key.

In setups where imparting key information can happen in a secure way, secret key cryptography is used. Public key cryptography is supposed to make secret key cryptography more secure and is used where such a need exists.

The message encryption functionality uses the library cryptix.jar, provided by Cryptix, for generating keys and for encryption. This file comes bundled with the FioranoMQ installation. It can be found in the FIORANO_HOME%/extlib/cryptix directory of the FioranoMQ installation.

Message Encryption Characteristics

FioranoMQ provides message encryption on a per message as well as on a per destination basis.

In per message encryption, clients can enable or disable encryption for every message. In per message encryption, encryption is done by a client before sending data on to the network. Decryption must be performed by the receiver client application before dealing with the message.

In per destination encryption, all messages sent to a particular destination (topic or queue) are encrypted, thus providing a secured channel. A destination is marked as encrypted at the time of its creation. All messages would be delivered decrypted to subscribing applications. Thus, a client application does not have to explicitly decrypt a received message.

Encryption involves only encrypting the payload of the message and not its JMS header. This allows usage of the same set of APIs associated with message headers as well as message selectors, irrespective of whether message encryption is enabled or not.

Chapter 13: Message Compression

Message compression is a functionality by which messages that are sent through FioranoMQ can be compressed while sending them and are restored to their original size before delivering them to the message consumers.

Compression has the advantage of improving performance. Less bandwidth is used during message transfer. Memory and storage requirements on the server are reduced. This functionality is important for performance-sensitive applications that operate over WAN links. In addition, Fiorano extends compression support for server-to-server communication.

Base Implementation

Many data compression implementations have been developed in the past, out of which the Zlib implementation is the most significant one. This implementation is based on "Zlib Compressed Data Format Specification Version 3.3".

This specification defines a lossless compressed data format. The advantages of this compression implementation as per the specification are:

- Is independent of CPU type, operating system, file system and character set. Hence can be used for interchange.
- Can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage. Hence can be used in data communications.
- Can be implemented readily in a manner not covered by patents. Hence can be practiced freely.
- Can use a number of different compression methods.

In FioranoMQ, the Zlib implementation provided by Sun in the default Java runtime library (java.util.zip) has been used. This implementation provides the 'deflate' and 'inflate' mechanisms using different compression levels and compression strategies. Compression level is the amount of compression required. Compression strategy is the actual compression method used. The default strategy uses a combination of LZ77 algorithm and Huffman coding.

Message Compression Characteristics

FioranoMQ provides message compression on per message as well as on a per destination basis. In per message compression, clients can enable or disable compression for every message. In per destination compression, all messages sent to a particular destination (topic or queue) are compressed.

Client applications can choose the compression level and strategy from the ones provided by the Zlib specifications using public APIs.

The significant compression levels are:

- NO_COMPRESSION
- BEST_SPEED (fastest compression)
- BEST_COMPRESSION
- DEFAULT_COMPRESSION

There are ten possible compression levels (0-9) available to choose from, where BEST_SPEED is defined as 1 and BEST_COMPRESSION is defined as 9.

The possible values for compression strategy are:

FILTERED: Compression strategy best used for data consisting primarily of small values with a slightly random distribution. It forces more of Huffman coding and less string matching.

HUFFMAN_ONLY

DEFAULT_STRATEGY: This uses a combination of LZ77 method and Huffman coding.

The compression support provided helps a client application to decide on the optimum compression level and compression strategy by providing APIs to check the compression ratio achieved after a message is sent or received.

Compression involves only compressing the payload of the message and not its JMS header. This allows usage of the same set of APIs associated with message headers as well as message selectors, irrespective of whether message compression is enabled or not.

FioranoMQ's implementation also allows users to plug in their proprietary compression implementation, which would override the default implementation.

Chapter 14: FioranoMQ Clustering

In real world applications of messaging servers, it is a common requirement to manage a heavy load of connections. In such cases, it is not just sufficient to have the messaging server installed on the best of hardware available in the market. There must be a more suitable, economical and scalable approach to the solution of this problem.

The most appropriate solution is to have 'n' number of messaging servers communicate with each other, share the load between them and work well in synchronization. A logical unit consisting of these 'n' servers (and some other software components) is called a cluster. A cluster also provides support for fail over, which is not feasible with a single server.

Common problems of Real-world Systems

This section gives an introduction on the common problems in distributed systems in the real world.

Client unable to connect

A server may be only temporarily unable to accept a connection request. An example could be socket buffer (backlog) that fills up if too many clients try to connect at the same time. The most desirable behavior for the client is to transparently attempt some reconnections first.

Since the server may be completely down rather than temporarily overloaded, the client needs to be able to connect to alternate backup servers. If this list of backup servers is retrieved as a parameter of a connectionfactory object, the client code may become non-portable.

Connection to the server is lost

This again is an important failure that should be handled. Client side persistence should exist. This store and forward feature enables a client to operate in disconnected mode and avoid loss of messages. A seamless integration of client-side persistence should be transparent, should allow for transacted sessions and should cater for duplicate messages.

The server runs out of resources

There are many resources that may effectively render a server inaccessible because of their shortage: connections, RAM, disk space, threads, file descriptors, sockets and possibly others. A cluster of servers can provide more resources, allow for distributing requests more adequately (load-balancing) and configure servers as standby, ready to take over in case of emergency. To preserve application portability, the cluster should appear as single (super) server, i.e., load balancing and failover should be transparent to clients and combinable with each other. In some cases, the shortage of a resource may be only temporary and hence it may be advisable for a client to first try to reconnect for some time until giving up and looking for an alternate server. Again, such an option should be combinable with load balancing.

The server goes down altogether

If a server crashes, the clients connected to it need to be able to continue by connecting to a secondary server. This scenario is termed fail over. Once a server recovers, it needs to be reactivated for taking over the tasks assigned to it thereby restoring the state before the crash. It is termed hot fail over if processing can continue seamlessly (with nearly no latency). This requires a secondary server to be up and running and to have access to persistent state and message data.

FioranoMQ: The solution

FioranoMQ provides the following features to solve the above problems:

Automatic Fail over protection

Failure of a component should not cause failure of the whole system. So backup of important components is required to provide high availability. There should not be a single point of failure. FioranoMQ's runtime invocation ensures that clients connected to a server are automatically failed over to the back up server. When an application wants to connect to a particular server, N attempts are made to connect to this server. If connection still fails, the runtime library tries the URL of backup servers specified in the connection factory being used to open the connection. Since this operation is performed automatically by Fiorano's runtime invocation, the application is not required to fetch the list of backup URLs. This avoids any vendor lock-in as the application code is complete JMS.

Transparency and code portability

The automatic fail over protection mechanism should be completely transparent to the client. FioranoMQ provides this transparency by making the reconnection mechanism invisible to the client. The reconnection is done automatically by the runtime library. This is achieved by the server configuration and does not require use of any proprietary APIs and hence the client code is completely portable across any other JMS server.

Configurability

FioranoMQ achieves a very high degree of configurability through its modular design. Interfaces for a number of modules are available to the public, which allow a developer to implement their own version of the module and plug it in. For example, a developer is free to write a version of Log Module that displays information in a Java Frame instead of the java console and plug it in. Besides these modules, a number of configurable flags are provided to the developer through the server's config file, which allows a developer to tweak various parameters of the server. For example, FioranoMQ provides a configurable option for the maximum number of clients that are waiting to connect to the server. This figure defaults to 500. This figure can be increased or decreased through the server's configuration file. Note that this figure represents the length of the queue of pending sockets and has no relationship with the maximum number of simultaneous clients that FioranoMQ can support.

Admin system

Availability of the system holding the administered objects (admin system) must be provided in the first place. Admin system should be different from the client systems and it should be backed up by some kind of a secondary. Client access to primary or backup admin system should be transparent. FioranoMQ allows the administrator to configure the admin system storage as required.

The administrator can choose the storage media of administered objects from the following options:

- JNDI compliant Directory Server
- RDBMS Server
- XML File
- Fiorano's proprietary file format (that also uses JNDI)

Besides these available options, a developer is free to write his own customized version of admin system. Storing the admin objects in a central Directory Server, allows a client to directly lookup these objects through JNDI. This doesn't require them to go through the FioranoMQ server (though that is also allowed). This makes the admin system totally independent of the server.

Connection to the server is lost

FioranoMQ provides a client with the ability to automatically connect to a fail over or backup FioranoMQ server. This mechanism works if the client's existing connection breaks

Or the client's primary server is unavailable at the time of creation of the connection

This mechanism is transparent to the client application. In case an existing connection breaks and there is no backup URL specified, the client application can continue its routine operations of pushing more messages. These messages are stored in a local repository on the client machine. Meanwhile, the runtime library automatically tries to re-connect back to the server behind the scenes periodically. Once the connection has been established, it automatically transfers all the messages stored in the local repository to the server. This provides the client application with store and forward facility for a publisher off the shelf. More importantly, this does not require the use of any proprietary APIs and hence avoids vendor lock-in of any kind.

Server runs out of resources

FioranoMQ's dispatcher is the load-balancing component within the Fiorano server. Dispatcher can be turned on or off easily through the server's configuration file. If enabled, the server distributes incoming connections to members of its clusters. Again, this does not require special APIs at the client application. The application only sees the dispatcher-enabled server. The dispatcher administrator is free to add/remove members in the dispatcher cluster any time. If a member server goes down in the cluster due to any reason, all the applications connected to this server shift to one of the other members in the cluster. FioranoMQ also provides fail over for the dispatcher server, which means that users can setup a secondary dispatcher that is used in cases where the dispatcher server goes down. The client system can have the URL of the secondary dispatcher as its backup URL and when the primary dispatcher goes down the client system is automatically load-balanced by the secondary dispatcher amongst the same member servers. An important requirement for running FioranoMQ servers in a cluster is that the TopicConnectionFactories (TCF), Topics, QueueConnectionFactories (QCF), Queues and UnifiedConnectionFactories (CF) should exist on all servers running in the clustered environment and on the server acting as a dispatcher. These components are used by the clients to make connections to the least loaded FioranoMQ server through the dispatcher. The TCFs, QCFs and CFs replication amongst servers is required to make the new connections to the server. The Topic and Queue replication is required for automatic fail over support for clients. This implies that if the client connection to any of the servers in the cluster goes down, the client gets connected to another server running in this cluster.

Server's connection to a client is lost

If a connection with a consumer is broken due to any reason and if the consumer is durable, the server keeps storing the messages for this consumer published by the producer. These messages are made available to the consumer next time it logs into the system. Fiorano's runtime library also pings the connections to the server periodically (with a configurable time difference between two pings), which allows the server to detect dead sockets and clean them up. This becomes a lifesaver in case of network failure, which is not detected by the JVM unless a write operation is performed on it, which is usually not the case for a consumer.

Server-to-server communication

It is a common requirement of real world applications to allow clients to exchange information seamlessly across servers. The Repeater and Bridge components of FioranoMQ are used for server to-server communication over topics and queues respectively. Apart from FioranoMQ - FioranoMQ bridge, bridges are available for all other JMS servers such as IBM WebsphereMQ, MSMQ and Tibco Rendezvous etc.

Scalability

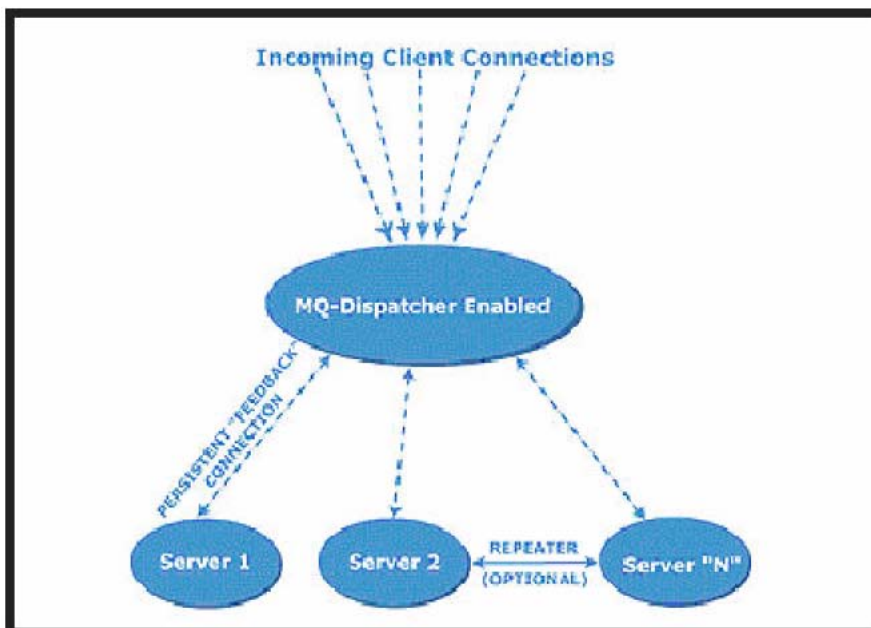
The load balancing and fail over protection architecture of FioranoMQ server allows unlimited scalability in terms of the number of client applications that can concurrently avail of JMS services. Thousands of concurrent client connections can be supported by a single cluster of servers. Combined with server-to-server communication, the Fiorano clustering architecture provides a very robust solution for a vast set of customer problems. For handling thousands of concurrent client connections, FioranoMQ also provides scalable connection management.

Clustering Components

FioranoMQ provides extensive clustering support using components such as dispatcher, repeater, and bridge. The dispatcher is used for load balancing (distributing load) client connections amongst different servers running in a clustered environment. The repeater and bridge are used for server-to-server communication over topics and queues, respectively. Apart from FioranoMQ - FioranoMQ bridge, bridges are available for all other JMS servers such as, IBM WebsphereMQ, MSMQ and Tibco Rendezvous etc. Detailed explanation of these components is provided in the subsequent sections of this chapter.

Dispatcher

FioranoMQ's load balancing architecture involves the use of a Dispatcher-enabled MQServer to route incoming client connections to the least-loaded server in a cluster, as illustrated in the Figure below.



The dispatcher component of FioranoMQ Server is typically connected to multiple FioranoMQ servers, which can run on different machines or on the same machine. All of these servers become part of the "cluster" that is serviced by the dispatcher.

FioranoMQ Dispatcher runs as part of a "server" process and maintains a persistent connection with each FioranoMQ server in its cluster. This persistent connection is used to pass information from the server to the dispatcher, enabling the dispatcher to maintain real-time "in-memory statistics" about the precise load in terms of the number of connections on each server. The dispatcher uses this information to determine the least loaded server in the cluster and accordingly route the new incoming client requests.

An advantage of using Fiorano Dispatcher is that no changes are required on the Client application to use Dispatcher. Once this functionality is turned on in a server, it automatically routes connection requests to the least loaded server. The server load is calculated internally by the dispatcher based on the maximum connections allowed on a particular server, and the number of active connections.

Preferred Server

At times, a given client application might want to connect to a particular server in a cluster. This can be done by setting a flag pointing to the "preferred server" within the cluster in the connection factory being used or in the lookup environment. The preferred server can be set through dispatcher configuration and is typically used by client applications that have previously created durable subscriptions on a particular FioranoMQ server within a known server cluster and wish to reconnect to the same server to retrieve messages.

Configuration Parameters

Parameter	Description
Login Name	Represents the login name used by the dispatcher to connect to MQ server member. The login should have admin privileges.
Password	Represents the password used by the dispatcher to connect to MQ server member.
AdminConnectionFactory	Specifies the admin connection factory used by the dispatcher to connect to MQ server member.
Server URL	Specifies the URL of the server in the cluster (Format: http://hostname:port) Server Admin URL specifies the admin URL of the server in the cluster (Format: http://hostname/port)
Backup Connect URL	URLs of the backup servers used in case primary server is down. Multiple backup URLs may be specifies as a semicolon-separated string of URLs. Example of a Backup Connect URL is http://backupServer1:1856; http://backupServer2:1856
MaxClientConnections	Specifies the weight associated with a member server of a cluster. A member server with MaxClientConnections set to 2 will allow twice the number of connections that can be created with member server with weight 1.
SecurityManager	The Security Manager implementation used to create secure connections with the MQ server. The manager should be an implementation of the **fiorano.jms.runtime** interface provided by FioranoMQ.
TransportProtocol (TCP/HTTP)	Protocol used for communicating with server. Transport protocol can be set to either TCP or HTTP
java.naming.security.protocol	Name of the security protocol used to create secure connections with the MQ server. The possible values that this variable can take are PHAOS_SSL and SUN_SSL

Repeater

FioranoMQ architecture allows multiple FioranoMQ Servers to be connected together, allowing clients connected on one server to exchange information with clients connected on any other MQ server. Using FioranoMQ Repeater, servers can be connected both over LAN (Local Area Networks) and WAN (Wide Area Networks)

FioranoMQ Server clustering allows clients connected on different FioranoMQ servers to exchange information by setting up an instance of the FioranoMQ Repeater. This feature is particularly useful in deploying applications that need to communicate with other applications across geographically distributed sites. For instance, consider an organization having offices in New York, San Francisco and Boston. In this case a Client of MQ Server located in Boston office can communicate with a client of MQ Server located in New York office. This is easily achievable with server-to-server communication, facilitated by repeater. With server-to-server communication, each client application in Boston only needs to connect to the Boston FioranoMQ server.

The FioranoMQ Administrator can configure the Repeater to automatically forward relevant messages from the Boston server to FioranoMQ server in New York and/or San Francisco, as per requirements. These messages are delivered to the subscriber applications that are connected to the New York and/or San Francisco servers. Moreover, in case, there are any transient network failures across the WAN connecting Boston, New York and San Francisco, none of the client applications are affected. Publishers can continue to publish messages locally. These messages are persistent on the local server of the publisher if a durable link (For more information, read the Subscription Mode and Choice of Selectors section) on the repeater connects the concerned FioranoMQ servers. The subscribers stay connected and receive messages, if any are available, from their local server. The repeater also takes care of reconnecting the servers in case of temporary network failures.

FioranoMQ Repeater enables the communication between different servers by using the Publish/Subscribe messaging model. This implies that information is exchanged between the topics on different servers and the repeater cannot be used for exchange of information over queues (For details on server-to-server communication over Queues, refer to the section "FioranoMQ Bridge: Integrated Message Queuing"). All server-to-server communication is handled transparently by FioranoMQ internally and the client application does not need to be modified in any way. MQ Servers can be part of the same LAN or can be spread across multiple WANs.

FioranoMQ Repeater allows information exchange over SSL/HTTP/HTTPS in addition to the default TCP/IP communication.

Salient Features

FioranoMQ Repeater offers the following features:

- Easy Configuration: FioranoMQ Repeater can run as embedded in the same container in which the server is running, or can run as a standalone component (separate process) and can be used to wire multiple servers. FioranoMQ Repeater provides complete power to the enterprise administrator to configure MQ Servers in any network topology. Administrator can set up topics on source and target servers, which exchanges information between them. Configuring the Repeater is simple and is XML based.
- Connection Topology: The Administrator of FioranoMQ can configure the Repeater to setup connection between servers and propagate messages on the connection.

The repeater can be configured to support different kinds of network topologies which are as follows:

Hub-spoke: A source server can be linked with N number of target servers and vice-versa. In the former case, the single source server acts like a message broadcast hub for all the target servers. All messages published on the source server can reach one or all of the target servers depending on the requirement. Similarly, in an opposite scenario, a single target server can act as a hub for many source servers and receive messages published on one or all the source servers.

Mesh: A cluster of FioranoMQ servers can be set up such that each server is connected to all the other servers in the cluster through the repeater, forming a mesh type of structure. In this case, messages published on any server can reach any other server in the cluster, depending on the topics on which the messages are published.

Bus based: The repeater can be used to set up a cluster of FioranoMQ servers, in which a single source server represents a message bus while many target servers behave as recipients of messages from this bus, in such a way that one message is received by one and only one target server at any point of time.

No changes in the client application

Client applications can communicate to any number of FioranoMQ servers that are connected by the Repeater. Clients connected to one server can exchange messages with clients connected on another server, without each client having to explicitly connect to the same server. The client application does not need to be modified in any manner - the FioranoMQ Repeater manages to forward messages pertaining to a Topic across servers. The Administrator has to only configure the repeater to forward messages pertaining to desired Topics across servers.

Robustness in handling network failures

Data transfers between multiple FioranoMQ servers (connected to each other through Repeater) can be optionally made to use Persistent Messages/DurableSubscriptions. In this case, messages transferred between servers are always logged to persistent storage, making the system highly reliable and robust in the event of network failure.

In case a network connection goes down, the repeater tries to bring it up. The repeater tries to reconnect to the server to which it has lost the connection, repeatedly, with a small interval between each try. This ping time interval is configurable through Fiorano Studio. The pinging operation continues until a connection is reestablished (when the "down" server finally comes up again)

Subscription Mode & Choice of Selectors

The FioranoMQ Administrator can configure the Repeater to create either Durable or Non-Durable link between the source and target servers. A durable link can be used to ensure that no messages are lost across the repeater in case of network failure.

Message selectors can additionally be set on a link between servers to allow only the required messages to be exchanged between them. These can be useful, especially in setting up the bus-based network topology. For more information, read the Connection Topology section.

Request/Reply Across Repeater

The Repeater provides functionality for using request-reply service over FioranoMQ servers, which are linked by repeater. This allows a requestor, publishing request messages on topic t1 on server S1 to get replies for the requests from a replier on topic t2 on server s2. The repeater can be set up effortlessly for such a request-reply scenario.

Dynamic Replication Links

FioranoMQ can be used to create new replication links dynamically. This enables the applications to replicate messages on topics that are created after the repeater has started. Administrators would not have to manually add replication links for all the topics. They can only specify a pattern like 'ABC*' which means that FioranoMQ repeater would create replication links for all the topics that matches the pattern. Also in case a new matching topic gets created, after the repeater has started, then it dynamically create replication link for that topic.

Repeater with Load Balancing

The repeater can be put its best use in a Load Balanced cluster of FioranoMQ servers. FioranoMQ uses the dispatcher for load balancing client connections amongst different FioranoMQ servers.

Repeater Link

The repeater replicates messages in the link specified between a source and a target server. A repeater can have N number of links configured. By default, the server sets up only a single link to the repeater. The properties can be edited related to this default link before creating and managing additional links in the online mode. The Link element within the Repeater Manager MBean provides/has the following information:

Status: Specifies the link is running or not.

Note: This is not a read-only parameter and its value can't be edited through any tool.

SourceServer: Specifies the server on which subscriptions are created Source Server contains the ConnectionInfo.

TargetServer: Specifies the Server on which publishers are created Target Server contains the ConnectionInfo.

Connection Info

The information present in this element represents the connection information that would enable the repeater to connect to source or target servers. Target Server as well as Source Server within a link is associated with an instance of Connection Info. An instance of ConnectionInfo contains the elements as defined in table defining Configuration parameters.

Link Topic Info

Each Link could be associated with one or more Topic Links. Each Topic Link refers to a source/target topic information. Repeater picks up messages from source topic and sends them to the target topic.

A link could also have an instance of Reqest/Reply Topic. Information regarding the source and target topics for request-reply services only. It is used when LinkTopic uses a request-reply type of message transfer.

Configurable parameters of LinkTopicInfo are as summarized in the table below.

Configuration Parameters

Topic Link Info Parameters	
SourceTopic Name	Specifies the name of the topic on which subscriptions are made on source server of the link, containing this LinkTopicInfo. This name supports wild character, '*', which if specified enables the repeater to create subscriptions on all the topics which matches the source Topic. For example, for source topic name 'ABC*', subscriptions will be made on topics like ABC1, ABC12, ABCDEF etc..
Target Topic Name	The name of the topic on which messages received on the above subscription are forwarded on target server of the link, containing this Link-TopicInfo.
ReplyOn	Specifies the topic name on which the repeater will listen for the replies, which it receives on the requests that it forwards, on this LinkTopicInfo.
isDurable	Specifies whether the link between the source and target is duarable. A durable link can be used to ensure that no messages are lost across the repeater in case of network failure. The possible values for this variable are "YES" and "NO".
Message Selector	Specifies the selector that is set on a link between servers to allow only the required messages to be exchanged between them.
Type	Specifies whether the link should be always connected to the target server or only replicate if a subscriber exists.
ReplyTopicInfo Parameters	
ReplyTopicName	Specifies the name of the ReplyTopic..
isDurable	Specifies whether the link between the source and target is durable. A durable link can be used to ensure that no messages are lost across the repeater in case of network failure. The possible values for this variable are "YES" and "NO".
Message Selector	Specifies the selector that is set on a link between servers to allow only the required messages to be exchanged between them.

Wild Character Support

FioranoMQ provides support for wild-card characters in repeater configuration so that separate links need not be added for each topic. A user can specify wildcard characters in the source topic. All topics starting with the string mentioned in the source topic can be repeated.

Repeater can be configured to replicate messages that match a particular pattern. The pattern can be specified in the source topic name in the Properies Name. For example, if the Source Topic Name is specified as "ABC*", the topics which match this pattern (all the topics starting with the string "ABC" on the source server) is repeated across two servers. Hence all the subscribers who have subscribed on ABC, ABC1, ABCZ and so on would be able to receive messages published on source topics ABC, ABC1 and ABCZ respectively through the FioranoMQ repeater.

The dynamically created topics which matches the pattern, 'ABC*' is also replicated. So in case 'ABC2' gets created after the repeater has started, then it dynamically creates a replication link for 'ABC2' (topic on source server) to 'ABC2'(topic on target server). Also in case a topic name (like 'ABD1') gets created which does not match the pattern ('ABC*'), then replication link for it would not be added.

Dynamic Link Propagation

Repeater can be configured to replicate messages only on demand, i.e. messages would travel from source to target only if there is a consumer (active or passive) on the target server interested in these messages. A pre-configured link in the repeater remains in "stopped" mode if there are no consumers on the destination. This link is activated as soon as a consumer is created. However, by default this feature is turned off.

Note: For this functionality to work events should be turned on in the server to which repeaters is connected to.

Request/Reply across Servers

The FioranoMQ Repeater provides a mechanism for using request-reply service across two servers. An intermediate Topic on target server is required to receive replies from the replier and forward them to the requestor.

Scenario: Let’s analyze the following situation:

A requestor (say REQ), connected to FMQServer Server1, is publishing request messages on topic t1. The requestor is then waiting for reply for the requests on a temporary topic, created for the connection.

A replier (say REP), connected to FMQServer Server2, is subscribing to request messages on topic t2. On receipt of messages, replier sends a reply to the request message on the reply Topic that is specified in the JMSReplyTo property of the request message.

Using repeater, the requestor REQ on topic T1 of Server1 can get replies for the requests it makes.

Specify the replytopicname as T3 using Fiorano Studio.

Create a topic T3 on the Server2 (target server)

Refresh the Repeater.

The SourceTopicName and the TargetTopicName can have the same name. Similarly, the replyOn value signifies the topic that is used by the target server for publishing replies for the messages. This is for the messages that are forwarded on this topic link, represented by this LinkTopicInfo. The ReplyTopicInfo represents the information about the replier topic link that is used in request-reply service across the servers.

In addition, a permanent topic t3 has to be established on the target server if the reply is to be sent to a temporary topic on the source server. This is because the repeater requires a topic on which it listens to the replies for the requests that it has forwarded.

Bridge

Due to lack of a standard communication protocol, every messaging vendor invariably uses his proprietary protocol such as TCP/IP, HTTP and Multicast for client-server communication. Lack of "interoperability" across multiple messaging vendors poses serious problems for communication across messaging systems. For instance, FioranoMQ applications cannot push or pop messages to IBM WebSphereMQ queues and vice versa. This lack of interoperability among message queuing systems can be a problem for businesses that merge and want to integrate information systems based on different message queuing systems. FioranoMQ solves this problem by "bridging" FioranoMQ with IBM MQ Series, MSMQ, Tibrv and all other JMS providers' products.

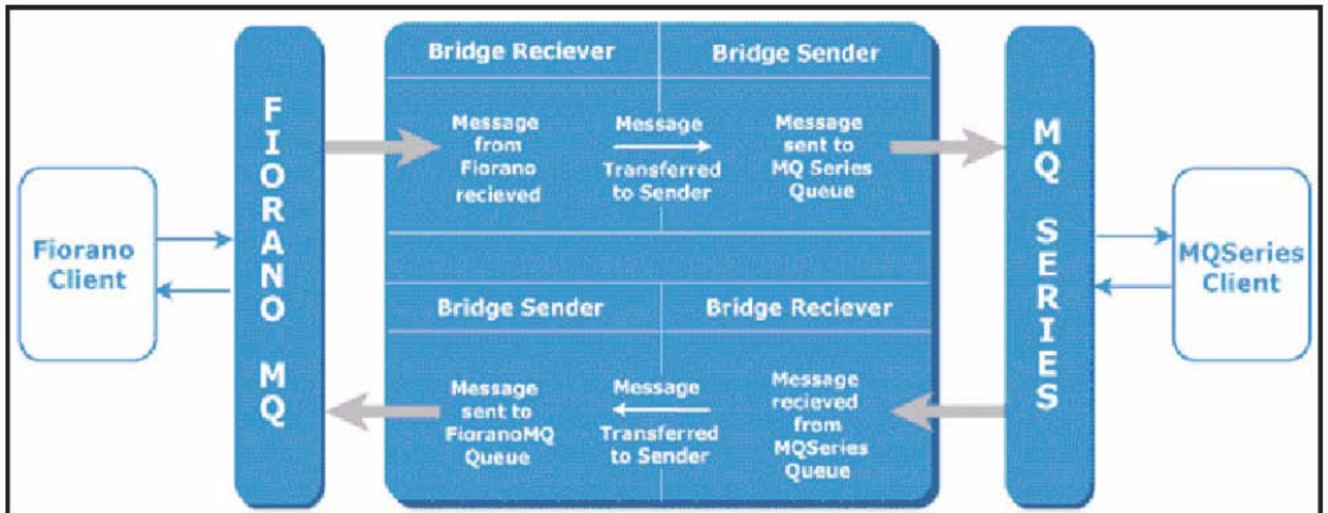
FioranoMQ Bridge solves the integration problem by allowing messages to be passed between FioranoMQ and other message queuing systems. For example, two banks merge and need to integrate their information systems. The management decides to consolidate all account information on IBM systems that use IBM WebSphereMQ for message exchange. In addition, management wants its customers to continue using their current ATM system. The ATM system receives requests for account information and dispatches requests to the server system using FioranoMQ messages. Instead of rewriting either application, the FioranoMQ Bridge can be used to forward requests and responses between the two different message queuing systems.

Bridge Architecture

The FioranoMQ Bridge is an open, standards based, 100% Java component. The FioranoMQ Bridge provides a set of standards based configurable services that allow message exchange between FioranoMQ and other message queuing systems.

Communication to any other JMS vendor is done using the standard JMS API. Communication to MSMQ is done using MSMQ Java APIs and communication to Tibrv is done using Tibrv Java APIs. The FioranoMQ Bridge sends messages as JMS Messages to the targeted JMS vendor. The FioranoMQ Bridge is configured using XML based configuration files that allow a single instance of the Bridge to "bridge" any number of messaging servers.

Messages are pushed to the remote system using the standard JMS API (or MSMQ/Tibrv APIs if the remote system is MSMQ/Tibrv). In addition, the Bridge creates receivers to asynchronously receive messages from the remote system. The Figure 8-3 depicts the flow between a FioranoMQ client and an IBM WebSphereMQ client through the FioranoMQ Bridge.



Forwarding Messages to Remote Queues

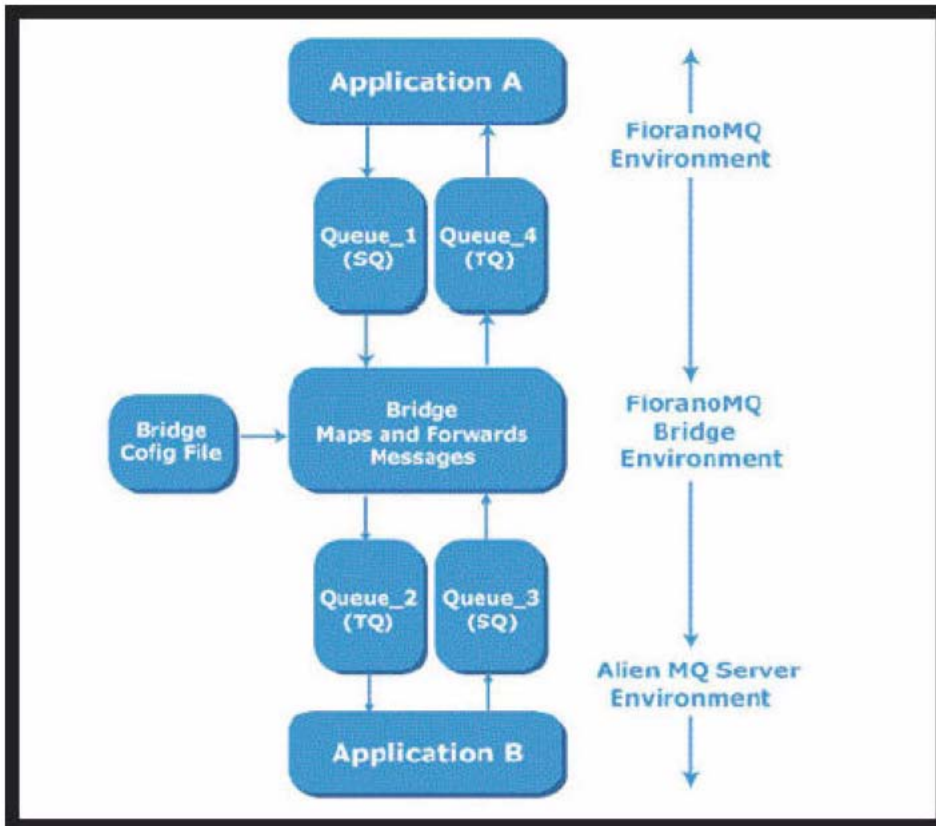
The FioranoMQ Bridge provides communication services between FioranoMQ and other messaging servers. As shown in the Figure 8-3, the Bridges create a relationship between a remote messaging server and FioranoMQ queues in order to send and receive messages between them.

This relationship is based on the use of two Bridges-specific entities, referred to as the Source Queue (SQ) and Target Queue (TQ):

Source Queue (SQ): An application always sends to a SQ, which is defined in the source messaging system. Messages received on a SQ are read by the Bridge and forwarded to the associated TQ. A SQ can either be a FioranoMQ queue or queue on other messaging server ("subject" if Tibrv).

Target Queue (TQ): This is the final target queue for sending a message. This queue is defined by the remote messaging system. A Bridge sends messages (retrieved from the associated SQ) to this queue for subsequent processing by the target application. A TQ can be either a FioranoMQ queue or queue on other messaging server ("subject" if Tibrv). The Figure below shows how the Bridge forwards a message from a FioranoMQ SQ to its associated Remote TQ.

For example, Application A is an ATM application that uses FioranoMQ for message exchange. It is designed to receive requests for account inquiries, send these requests to Application B, and return account information to bank customers. Application B is an account lookup application that uses IBM WebSphereMQ for message exchange.



Application A places an account inquiry message on Queue_1, a FioranoMQ queue. The part of FioranoMQ - WebSphereMQ Bridge reads the message on Queue_1. It maps the message header data into IBM WebSphereMQ format and forwards the message to Queue_2, an IBM WebSphereMQ queue. Application B reads the message on Queue_2, looks up the requested account information, and places the account information in a reply message. The message is placed on Queue_3, an IBM WebSphereMQ queue. The FioranoMQ Bridge reads the message on Queue_3. It maps the message header data into Fiorano MessageQ format and forwards the message to Queue_4, a FioranoMQ queue. Application A reads the message on Queue_4 and displays the account information to the customer.

Bridge Features

XML based: The FioranoMQ administrator can configure the FioranoMQ Bridge using an XML config file. FioranoMQ Bridge gives enterprise administrator complete power to configure message queuing servers in any network topology. The administrator can set up queues that need to be mapped between target and source servers. Configuring the Bridges is simple and is based on XML.

Robustness in the Presence of Network Failures: The Bridge has a built in feature to auto-connect with various messaging servers. In case a network connection or a messaging server goes down, the Bridge automatically tries to reconnect to the "down" servers after every "configurable" period of time. This pinging operation continues until a connection is re-established (when the "down" server finally comes up again).

Avoiding Loopback in Bridges: Cyclic links have been enabled in bridges. An FioranoMQ to FioranoMQ Bridge can result in an infinite loop being set up because of the bridge configuration. A check has been added in the bridge, which can disallow a message to enter in an infinite loop. This check is enforced only if a particular configuration flag is turned on using Fiorano Studio.

Logging and Trace Options: The FioranoMQ Bridge provides for comprehensive logging facilities. Logs can be redirected to files or console. The administrator can install "customized" logging mechanisms. Trace levels for the Bridge components can be set using Fiorano Studio.

Bridge Configuration

Configuring the Bridge consists of adding links and channels to the bridge and specifying the queue information to allow bi-directional communication across messaging servers.

In the offline mode, the administrator can easily add links to the bridge and configure the same to source and target servers for message replication. Cluster administrators are provided with a template configuration file. FioranoMQ installation provides a default ConfigswithBridges.xml file in the default FioranoMQ profile (fmq\server\profiles\FioranoMQ 9\conf directory of MQ installation). This provides default Bridge configuration, with two links added, linking the source and the target servers bi-directionally. This file needs to be renamed as configs.xml before starting the Fiorano Studio in the offline mode. The tool displays the Bridge with these default links that can be configured through offline tool.

The following table lists out the configurable Bridge Manager properties

Property Name Description

Parameter	**Description**
Avoid Loopback	This flag is used to avoid the loopback condition in bridge. The default value is true.
Name	Specifies the name of the Bridge
PingInterval	Specifies the time interval after which the bridge keeps pinging the source and target servers to reconnect after the connection goes down.

Link Properties

The bridge sends messages in the link specified between a source and a target server. A Bridge can have N number of links configured. By default, the server sets up only a single link to the Bridge. The properties can be edited related to this default link before creating and managing additional links or channels in the online mode. The Link element within the Bridge Manager MBean contains the following elements:

SourceServer

Specifies the server on which subscriptions are created. The Source Server element contains the ConnectionInfo.

TargetServer

Specifies the server on which publishers are created Target Server contains the ConnectionInfo.

Connection Info Properties

The information present in this element enables the Bridge to connect to servers even if they are behind Proxy or Firewall. It additionally enables secure connections by providing security certificates and the security protocol to be used. Connection Info comprises of ServerURL, UserName, Password, ProxyURL, and ServerSecurityManager. The type property of this element indicates the protocol over which the bridge would be connecting to the source or target server.

ConnectionInfo contains the following elements:

Parameter	Description
ServerType	Type of the Messaging Server to connect to. Currently only four server types are supported: JMS, MSMQ, IBM WebSphereMQ and Tibrv.
Parameter	These contain the different environment parameters required by the Connector to connect to the Source/Target Server. An important use of these parameters is for configuring the protocol and the connection factory over which the bridge will be establishing connections with the source and the target servers.

A channel can now be added to the existing link. The channel specifies the source and target queue information. A channel has the following information:

Parameter	Description
Name	Name of the Channel
SourceQueue	Specifies the source queue information required by the Bridge. This contains the SourceQueuename.
TargetQueue	Specifies the target queue information of the channel. This contains the TargetQueuename.
Parameter	These contain the different environment parameters required by the Connector to connect to the Source/Target Server. An important use of these parameters is for configuring the protocol and the connection factory over which the bridge will be establishing connections with the source and the target servers.
Type	Type of the Messaging Server to connect to. Currently the types that are supported include: JMS, MSMQ, IBM WebSphereMQ and Tibrv.

Chapter 15: Large Message Support

With Large Message Support (LMS) in FioranoMQ, clients can transfer large messages in the form of large files with theoretically no limit on the message size. Large messages can be attached with any JMS message and the client can be sure of a reliable and secure transfer of the message through FioranoMQ server.

Note: In this chapter, the terms "Large Message Support" and LMS is used interchangeably.

Salient Features

Reliable transfer of large messages

The large message sender and receiver use JMS semantics to transfer the JMS message containing the reference to large message that is to be transferred. When large message transfer is initiated, all fragments are sent through the server ensuring reliability. Files sizes running into Gigabytes can also be transferred.

No increase in cache/JVM heap size required

Although the message transfer happens through the server, the file fragments are sent to the receiver in a request reply fashion enabling the server to handle large message transfers without any increase in cache/JVM heap size.

The Large message transfer is not restricted to any queue or topic

As mentioned above, large message transfer conforms to JMS semantics, allowing the user to transfer the message on any queue or topic.

Resume functionality at both sender and receiver end

Broken transfers can be restored using the resume functionality built in the message class.

Message transfers are specific to the JMS Connection-users involved in the transfer. The user can resume the transfers at any later point provided the participant at the other end is available.

Minimal changes in the application code

All the message transfer properties are built into the message object while the message transfer semantics are built into the normal JMS send and receive calls. So, minimal code changes are needed to send and receive large sized data.

Using Fiorano LMS to transfer large files

A large message is defined as any JMS Message with a reference to a large sized file. Message fragmentation, reassembly, sequencing, duplication, and recovery are handled internally. When the large message is sent, it is only the reference to the large file that is sent with the JMS Message. The actual file transfer happens only after the receiver has received the JMS message containing the reference to large message and starts the saveTo operation on it.

Transferring a large message using Fiorano LMS involves the following phases:

- Message creation
- Starting the message transfer
- Tracking the message transfer
- Handling exceptions in the message transfer
- Resuming the message transfer

Message Creation

The large message is created using the complete JMS semantics. A JMS application can specify a string property; "JMSX_LM_PATH" to convert a JMS message to a large message. This string property specifies the absolute path of the large file that needs to be sent to the listening consumers. A consumer receives the large message just like any other JMS message.

Starting the message transfer

The large message transfer can be initialized by calling the normal JMS send call at the producer end. This posts the large message to the intended destination (queue/topic), which is received by the listening consumers. They receive this large message and can start the actual transfer of the large file by opting to save this message at the required location using the saveTo API.

Tracking the message transfer

Message transfer can be tracked asynchronously by registering a status listener with every large message. Status listener can be set using the setLMStatusListener API.

Handling exceptions during message transfer

Exceptions can be handled using the status listener registered with the large message. When an exception occurs while transferring the large message, the application is notified via the registered status listener. Applications can check the type of error and status of the transfer and may decide to resume/cancel the message transfer.

Resuming the message transfer

In case some failure occurs while transferring the message, then the application can resume the message transfer from the state at which the failure had occurred. The application can resume the message transfer at the two levels described below.

- Resume on exception: FioranoMQ notifies the application about the status of the message transfer using the registered status listener. If an exception occurs during message transfer, FioranoMQ notifies the application about the failure. On this failure notification, the application can opt to resume the message transfer. It can resume the transfer using the resumeSend () or resumeSaveTo () APIs provided in the message object.

- Resume on startup: The JMS Connection object keeps track of all active message transfers that were going on for that particular connection. In case the application becomes unavailable (JVM down) while taking part in the message transfer, it can then check the list of unfinished transfers from the connection object. The connection object provides list of unfinished messages that are required to be sent and received. The application can resume the transfer using the resumeSend () or resumeSaveTo () APIs provided in the message object.

Salient Features

The actual transfer of the large file attached with the large message follows certain message transfer semantics that is explained below

Consumer Discovery

The message producer waits for the initial message (handshake) from the consumer before any message fragments are sent. In case of point-to-point messaging model, the discovery phase ends as soon as the handshake message is received from the consumer. In case of publish subscribe model, the discovery phase continues to receive more handshake messages from new consumers. The duration for which the producer waits for handshake messages is determined by the requestTimeOut value set by the user in the message object.

Fragment size

The large file is broken down to fragments of size <fragment size> as set by the user in the message object. Fragments of size in multiples of TCP/IP window size are recommended for optimal performance.

Window size

Window size is the number of message fragments that is sent by the message producer before an acknowledgment is received from the message consumer. For a window size of say 50, 50 fragments of size <fragment size> is sent to the server. Then the producer stops sending new fragments until an acknowledgment is received from the message consumer. When the acknowledgment is received, the message transfer continues.

Sequencing

Every message fragment sent has a sequence number attached to it by the message producer. The message consumer expects the fragments to be received in the same sequence as the order in which they were sent. If the sequence is disturbed then the consumer re-requests for the fragments that could not be received sequentially. Producer on receiving such request starts sending the fragments from the sequence number mentioned in the request.

Handling Duplication

It may happen in rare cases that a message fragment is received more than once. In such cases, the message consumer simply ignores the duplicate (other than the first) fragments.

Handling lost fragments

The message consumer identifies lost fragments using the sequence number associated with each fragment. When a fragment was not received due to some reason, the consumer re-requests for the fragments, which could not be received sequentially. Producer on receiving such request starts sending the fragments from the sequence number mentioned in the request.

Optimizing large message transfer

The time involved in transferring a large message (of size M) is the sum of

Time involved in transferring N JMS Messages of size M/N.

Time involved in I/O operations (reading the large file on the sender side + saving the large file at the receiver side).

Time involved in establishing the connection and ensuring fragment sequence and reliability.

Following factors can affect performance of LMS:

Fragment size

Small fragment size implies frequent I/O operations (reading the source file and saving the target file) and frequent socket calls to send the fragments to the server. Large fragment size implies increased memory usage both at the client end and server end. Optimal fragment size should be just small enough to be held in memory at the sender, server and receiver ends. Fragment size, whether small or large, should be in multiples of TCP/IP packet size for optimal performance.

Window size

Large window size implies increased memory usage at server side because of the possibility of receiver being slow to receive the messages from the server. After every window of fragments is sent, the message producer halts for an acknowledgment from the server. It might happen that receiver may not receive a fragment sequentially and request the fragments to be sent again. In such cases it is wise to have smaller window sizes when the expected rate of fragment loss is high. Another factor to be considered is the latency between the sender and the receiver. It is better to have a large window size when the latency is high.

Status message frequency

The setLMStatusListener call registers a status listener with the message transfer and informs the user application of the status of the message transfer. It also takes another parameter for the frequency of status updates required by the user. This value is the number of windows of message fragments that is sent before the user application is updated of the transfer status. A small value (say 1, for update after every window) hampers the message transfer ratei considerably. Also doing intensive operations in the 'onLMStatus', call takes up considerable time from the message transfer thread.

Known Limitations

- Only one file can be associated with a JMS message
- Large messages cannot be sent in a transacted session
- Resume API does not work well in some cases in case of publish-subscribe
- Resume directory cannot be shared by multiple consumers who are receiving the large message
- LMS does not work if AllowDurableConnections is set to TRUE
- LMS does not work in case of HA
- An instance of large message cannot be sent again unless the previous transfer using the same instance has completed
- LMS works only for unified connections provided by JMS 1.1 specifications.

Chapter 16: High Availability

Today's real-time enterprise solutions often deploy a messaging middleware that enables communication between various sub components. This middleware is entrusted with important data that should be delivered reliably and as fast as possible to the recipient application. The middleware server might also be required to store this data in its data store until it is picked up.

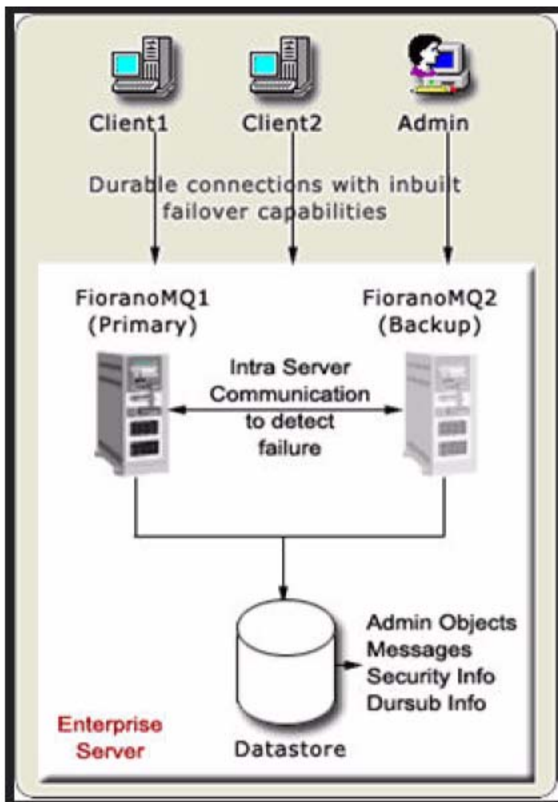
A failure of this middleware message bus might bring the entire system to its knees within seconds. Hence it is absolutely imperative for the messaging backbone to provide its backup, which allows messaging operations to resume quickly in the event of a failure of the running server. This backup server should restore the state prior to failure of the original message server. Any data that was stored previously in the server's data store should be accessible through this backup server and most importantly this operation of shifting from one server to its back up should be automatic and transparent to the client application.

FioranoMQ introduces High Availability (HA), which allows its applications to take advantage of its in-built fault tolerance capabilities. This guide discusses the salient features of FioranoMQ's HA solution. It explains the working and the underlying architecture of the entire solution. It also provides step-by-step instructions on enabling HA in FioranoMQ.

FioranoMQ's HA - An overview

FioranoMQ server when run in High Availability mode has a designated backup server, which is started along with the primary FioranoMQ server. In case the primary server becomes unavailable due to any reason, the backup server picks up all the messaging traffic immediately. This pair of primary and its backup server is known as an Enterprise Server and would be used to describe this pair throughout the document.

This Enterprise Server represents a Highly Available entity that appears as a single FioranoMQ server to its applications. JMS applications, during initialization, connect to the primary FioranoMQ server, if available. If the primary server goes down due to any reason, all connections are automatically routed to the backup server and communications are restored immediately. Since all this is transparent to the client application, it need not worry about the reconnection logic in its code as it is handled by FioranoMQ's runtime invocation internally.



HA Components

The section below describes the components and some of the associated concepts that together make up an Enterprise Server.

Backup Server

Fiorano's HA solution requires running a backup FioranoMQ server. This server (also referred to as secondary server in this document) can be started on the same or different physical machine. This server takes up all the messaging traffic immediately as soon as it detects unavailability of its peer.

Server States

FioranoMQ Servers that make up the Enterprise Server can be in either active or passive State. Active state refers to the normal working mode of the server. In passive mode, the server only monitors its peer and does not handle any client requests. Any client connection to the server in passive mode is refused. Upon startup, the server would establish communication with its peer server and upon finding it alive, enter into passive mode. It leaves its passive mode and becomes active (accepts client connections) only when it detects that its peer is unavailable due to any reason.

Intra-Enterprise Server Communication

Both Primary and Secondary server open and listen on a TCP/IP port that allows them to establish a dedicated connection between themselves. This port is different from the one used by client applications for connecting to the server and hence does not affect normal messaging operations going on in the active server. This connection is established during the initialization phase of the servers and is used for exchanging health and state information between the two servers. This information is used by the servers to switch state to Active from Passive if required.

Common Persistent Message Store

Both Primary and Secondary server would be required to logically access a common persistent message store. In order to achieve this, FioranoMQ Administrator can either point both the servers to the same physical database or can set up replication between the databases of the two servers. Both these options are available with Fiorano's file-based database as well as on any third party JDBC compliant RDBMS Server.

Common Admin and Security

Besides the message store, Primary and Backup Server in an enterprise server would be required to share the admin objects (Destinations and Connection factories) and Security Information (ACLs and User Information) amongst themselves. This is achieved by using a common naming and realm storage (RDBMS, LDAP etc.) or setting up replication on these databases between the two servers.

Gateway Machine

Consider a scenario in which the enterprise server consists of FioranoMQ server 1 and FioranoMQ server 2. Both these servers are constantly monitoring each other's status without any problems. Now, assume FioranoMQ server 2 (due to some network failure) goes out of network. Though FioranoMQ server 2 is still running but it is no longer connected to the network (and hence not accessible to FioranoMQ Server 2 and to client applications).

In this scenario, a third gateway machine is used to detect the HA server which is no longer available on the network. It becomes imperative to choose the gateway machine which itself is least expected to be out of the network. It makes sense to use the actual gateway server of the network in which the enterprise server is deployed as the Gateway machine for HA.

Note: In case the Gateway machine itself goes out of network, then HA continues to function properly as long as the two HA servers are present on the network. If one of the HA servers also goes out of the network, then it is not possible to reach the expected state. In this case, both the servers would switch to passive mode and the enterprise server is not be in a position to process any client request. However it would again be available for client requests when either HA server 2 or the gateway machine comes back in the network

FioranoMQ HA Salient Features

Shared and Replication database

FioranoMQ provides complete flexibility to administrators giving them an option to either use a shared database (between active and passive server) or use database replication (from active to passive server). So in scenarios where it is not possible to share the database, administrators can still use FioranoMQ's High Availability using the inbuilt replication support.

Application Fail over

In case the primary server becomes unavailable, all the client applications connected to it are automatically re-connected to the secondary server. The process of shifting from the primary server to the backup server or vice-versa is transparent to the application. It need not bother about writing reconnect logic in its code. This is achieved by connecting to the server through a Durable Connection. In case a backup server is available, the Durable Connection would connect to the backup server. Otherwise it waits for the server to restart. Further, it stores all the data sent during the disconnected period in a local repository and transfers this data as soon as the connection is re-established, thus making the system highly reliable and robust even in the case of network failures.

Note: Durable connections are a proprietary feature of FioranoMQ (though it does not require any proprietary APIs) and should not be confused with Durable Subscribers.

Data Store Consistency maintained between server switches

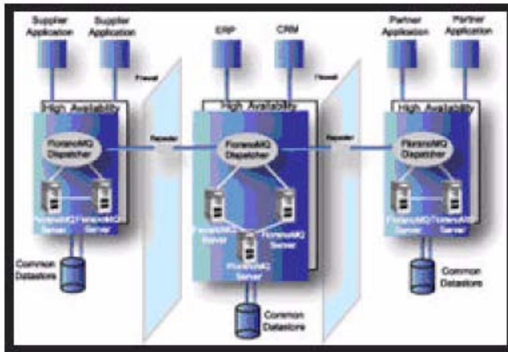
When the primary server becomes unavailable, its backend database state is conserved. This state is picked up by the secondary server when it becomes ready for action. This avoids loss of persistent information between server switches and at the same time provides access to the information that was stored through one server from its backup server. For example, all the messages that were published on to various destinations residing on the primary server before it went down are available to valid consumers coming through the secondary backup server without any message getting lost.

Expensive HA Hardware Not Required

Fiorano's HA solution is purely implemented in software and is not dependent on expensive hardware solutions. It can be run on any java-supported platform. With the shared database option, one might want to use RAID or SAN disks (if using HA over Fiorano's proprietary file-based data store) for enhanced speed and stability, but this hardware is not an essential component of Fiorano's HA solution. This hardware can also be avoided by using either replication support or using a central RDBMS server as the message store in the Enterprise Server.

Implementing a Cluster

The Enterprise Server can be clustered with other Enterprise Servers or even stand-alone FioranoMQ servers. This cluster together can share destinations (using a common naming store) and provide load-balancing facilities.



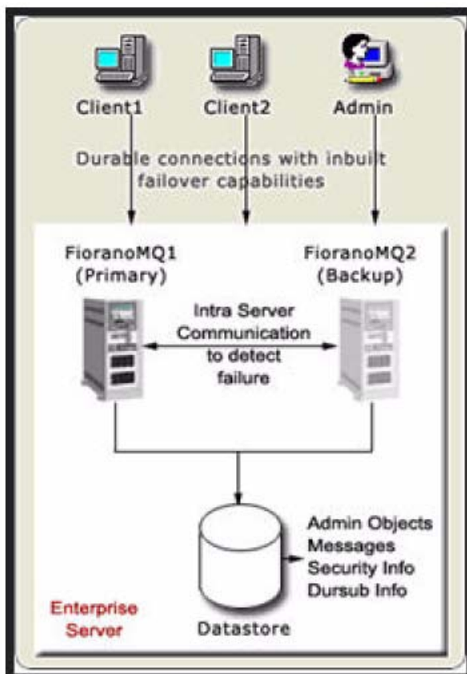
HA Example Scenario

This section of the document provides a description of events that occur in case one of the servers in the Enterprise Server becomes unavailable.

State - 1 (Normal Operation State)

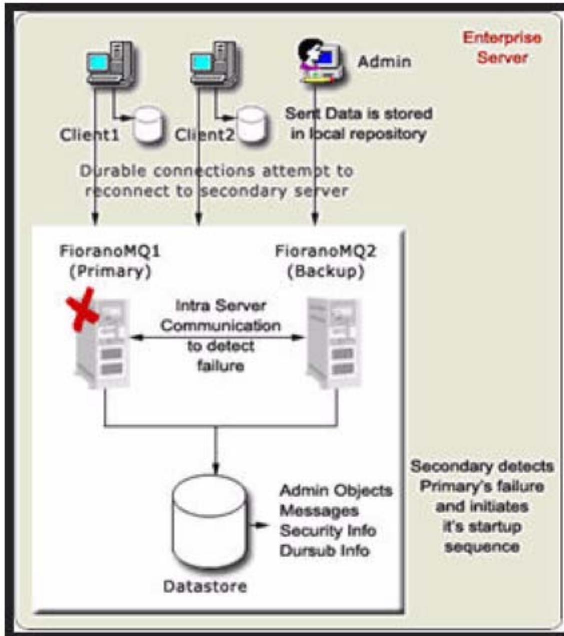
All client applications are connected to one of the servers in the Enterprise Server; Assume they are connected to the primary server at this instant. The backup server is up and running but is in passive mode. This server would not accept any client connections at this point of time. Primary and Backup Server are continuously exchanging health information over a dedicated channel. All persistent information is being stored in the back end data store through the primary server.

This is pictorially shown below:



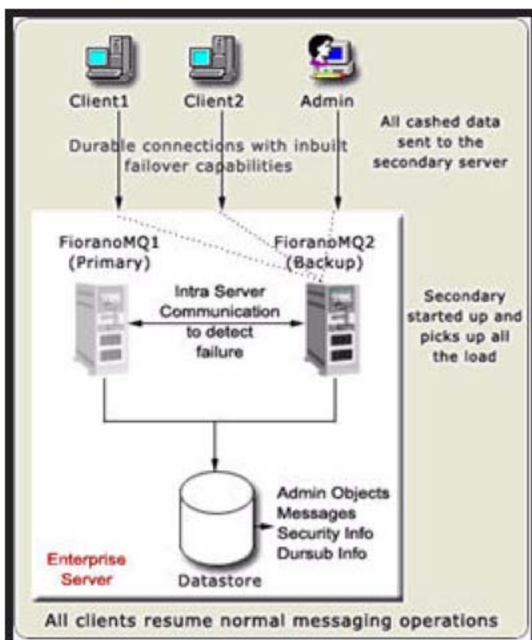
State - 2 (Active Server goes down)

The backup server detects the primary server's failure and initiates its start up sequence. All client applications connected to the primary server also detect the problem and Fiorano's runtime library internally attempts to re-connect back to the secondary server. New messages that are published in this downtime are stored in a local repository at all client machines.



State - 3 (Backup Server resumes operations)

The backup Server starts up completely and is ready to take up client connections. All client applications reconnect with the secondary server. The messages that were stored in the local repository (that were sent during down-time) are sent to the secondary server. All durable consumers pick up messages from where they had left.



Upon restarting the primary server, it would detect that its backup is alive and enter into passive mode. It would continuously ping the backup server and initiates its startup sequence if the backup server goes down due to any reason.

Limitations of HA

Client level transactions do not span across servers in the Enterprise Server when running in shared mode. Transacted sessions involving receivers would be rolled back in case the primary server crashes. Hence, the messages delivered in that transaction is redelivered to the receivers when connected to the backup server.

Distributed transactions, which are in execution during transition phase, becomes "in-doubt transactions" when the primary server goes down. These transactions get rolled back and can be recovered after the client reconnects to the secondary server.

JMS Topic Requestor may not receive its intended reply if failover occurs after a request is sent. This occurs because JMS Topic Requestor creates a non-durable subscriber, which can miss a message during failover. However, if a topic requestor creates a durable subscriber to listen for replies, then it works fine even during failover.

In case both HA servers (primary as well as backup) go down, the requestor receives a duplicate reply (with reDelivered Flag = true) for the first request made after fail over.

Chapter 17: Distributed Transactions

Communication between diverse applications is an essential requirement in today's distributed network environments. This communication can be at a single tier level such as, one application talking to the other in a single transaction, or at a multiple tier level such as, many applications talking to each other in one large transaction. The latter type, ability of a transaction to span multiple applications, is the fundamental issue in the context of multi-process communication.

The availability of low-cost computing power and increased network bandwidth gives rise to distributed component-based computing applications. Distributed computing applications and distributed transactions hold a promise for developing computing components for multi-tier applications, which can run on different platforms or through networks. In such a world, a transaction is defined as a unit of work composed of sets of operations on objects. A distributed component-based application is a configuration of services provided by different application components. These components are executed on physically independent systems running on multiple machines. To the user, these components appear as a single application running on a single physical machine.

Introduction

In the world of distributed computing and distributed transactions, a transaction could be defined as a group of statements representing a unit of work composed of a set of operations on objects. These groups of statements must be executed as a unit in totality. Elaborating on the same, transactions could also be viewed as sequences of operations on resources, such as read, write, or update, which transforms one consistent state of the system into a new consistent state. The basis of these transactions lies in the fundamental concept of an all-or-nothing proposition. Either all steps of a transaction must complete successfully or none of them should be completed.

In a large network, spread over multiple machines and involving many individual steps for the transaction, it is highly probable that one of those steps is not completed. This may occur due to many reasons such as flawed application logic, server failure, hardware failure, and network interruptions. Due to such unpredictable environmental factors, transactions must adhere to the following properties.

Atomicity: This is the all-or-nothing property. Either the entire sequence of operations is successful or unsuccessful. A transaction should be treated as a single unit of operation. Only completed transactions are committed and incomplete transactions are rolled back or restored to the state where it started. There is absolutely no possibility of partial work being committed.

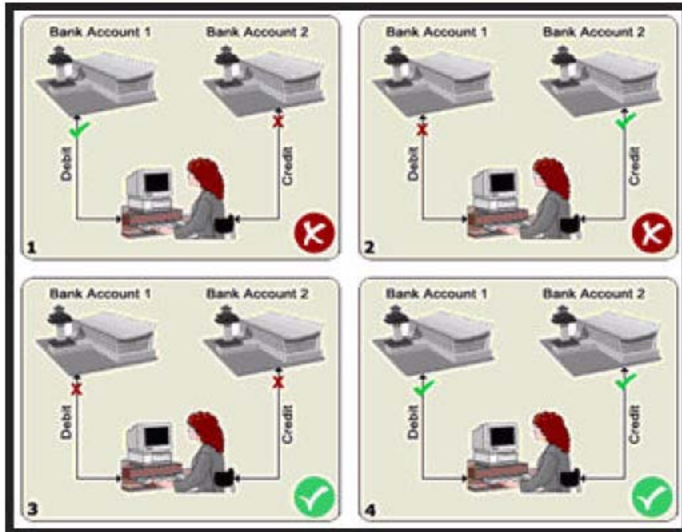
Consistency: A transaction maps one consistent state of the resources (for example, database) to another. Consistency is concerned with correctly reflecting the reality of the state of the resources. Some of the concrete examples of consistency are referential integrity of the database and unique primary keys in tables.

Isolation: A transaction should not reveal its results to other concurrent transactions before it commits. Isolation assures that transactions do not access data that is being concurrently updated. The other name for isolation is serialization.

Durability: Results of completed transactions have to be made permanent and cannot be erased from the database if the system fails. Resource managers ensure that the results of a transaction are not altered due to system failures.

Use Case

A familiar example of distributed transactions is transferring money from one bank account to another. The transfer comprises of two separate actions involving debit of a certain sum of dollars, and the credit of this sum to another account. Both these steps would be parts of a single transaction. Here, in this bank payment example, both the mentioned steps should be completed for the completion of the entire process. In case if one of them is completed and the other is not, the following possible problems might arise:



Case 1 and 2 are summarized in the following table.

Step 1	Step 2	Result
Completed successfully	Incomplete – Connection failure	Now, the amount has been debited from the customer's account but the payment has not been received. This causes the customer to lose money.
Payment has not yet been made by the customer.	Amount is credited to the seller's account	The bank loses money.

For these reasons, it makes sense to define a single transaction where the steps are set to ensure that the either-or-none proposition holds true. This would maintain that either both the steps are completed or none of them is completed giving some credence to the overall system and preventing any unintended losses to any concerned party.

Transactions do not always involve the transfer of funds as in the banking example just covered. Transactions are necessary for all kinds of business activities. For example, an online bookstore needs transactions to perform many activities such as ordering books from suppliers, transferring inventory from suppliers, updating available quantities of books accurately, charging customers appropriately for purchases, and fulfilling customer orders. All of these actions, and a multitude of others, may need to be executed within a transaction.

In other words, a transaction is a unit of work performed on behalf of a single client, delimiting a related set of operations and providing scope for a concurrency tool. Transactions guarantee consistency of shared state in the face of concurrent access by isolating one client's work from another and undoing a client's work in case of a failure.

Transactions and DTP System

A distributed transaction-processing (DTP) system defines and coordinates interactions between multiple users and databases (or other shared resources) residing on multiple machines. When a transaction includes operations in several databases or other shared resources, the goal of a DTP system is to carry out this transaction in an efficient, reliable, and coordinated way, and must comply with the above-mentioned ACID properties as far as possible.

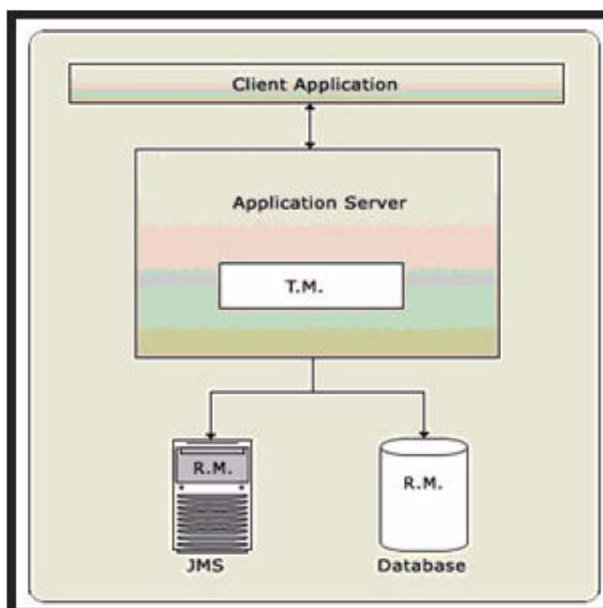
Components of a Distributed Transaction

The transaction manager and the resource manager are the two key elements of any transactional system. Generally, distributed transactions have five components.

They are:

- Transaction manager
- Application server
- Resource manager
- Application program
- Communication resource manager

Each of these contributes to the distributed transaction processing system by implementing different sets of transaction APIs and functionalities.



A transaction manager provides the services and management functions required to support transaction demarcation, transactional resource management, synchronization, and transaction context propagation.

An application server provides the infrastructure required to support the application runtime environment, which includes transaction state management. An example of such an application server is an EJB server.

A resource manager (RM) provides the application, access to resources through a resource adapter. The resource manager participates in distributed transactions by implementing a transaction resource interface. This interface is used by the transaction manager to communicate transaction association, transaction completion, and recovery work. FioranoMQ is a resource manager.

A transactional application specifies actions that form part of a transaction. These programs may require actions such as updating a database or sending messages. Such standalone Java client programs might want to control their transaction boundaries using a high-level interface provided by an application server or the transaction manager.

A communication resource manager (CRM) supports transaction context propagation and access to the transaction service for incoming requests.

From the transaction manager's perspective, the actual implementation of the transaction services does not need to be exposed. Only high-level interfaces need to be defined to allow transaction demarcation, resource enlistment, synchronization, and recovery process to be driven by the users of the transaction services.

FioranoMQ as a Distributed Transaction Resource Manager

The JMS specification provides a model that outlines how a messaging system should behave in a transactional environment. In the JMS transactional model, message producers and message consumers never participate in a single distributed transaction. Instead, the JMS specification has its own loosely coupled transaction model, whereby each message producer or message consumer has its own private transactional session with the messaging system, such as FioranoMQ.

This is due to the inherent nature of applications using messaging to communicate in an asynchronous environment. Senders and receivers of messages are abstractly decoupled from each other. A receiver may not be available at the time the sender initiates a transaction. Thus, the sender has a contract with the JMS messaging system, that messages produced within a transactional session are "committed" to the JMS messaging system in an all-or-nothing manner. Likewise, the receiver has another contract with the JMS provider that all messages being consumed within a transactional session are received in an all-or-nothing manner.

FioranoMQ, being standards based JMS server, implements the above-mentioned model to provide XA support in a Distributed Transaction environment using the JMS XA Service Provider Interface (JMS XA SPI).

In addition to the high performance "file based" data store, FioranoMQ has support for using a Relational Database (like Oracle) to be used as the message data store.

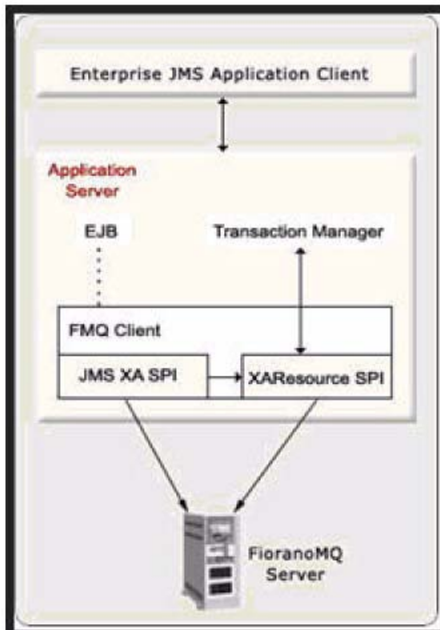
Transactions with J2EE

J2EE Applications include components that avail of the infrastructural services provided by the J2EE Container and Server, and therefore only need to focus on "Business logic".

Transactional support is an important infrastructural service offered by the J2EE platform. The specification describes the Java Transaction API (JTA), whose major interfaces include the javax.transaction. UserTransaction and the javax.jms. TransactionManager. The UserTransaction is exposed to the application components, while the underlying interaction between the J2EE server and the JTA TransactionManager is transparent to the application components. The JTA UserTransaction and JDBC's transactional support are both available to J2EE components.

An EJB or an Enterprise Java Bean running in an Application Server exposes the business logic methods that clients invoke to perform useful operations, such as depositing or withdrawing from a bank account. Enterprise beans are capable of handling transactions. This implies that EJBs can fully leverage the ACID properties to perform reliable, robust server side operations. Thus, EJBs are ideal modules for performing mission critical transactional tasks.

In an EJB, the code never interacts with the low level transactional system. The beans never interact with a transaction manager or a resource manager.



The application logic is required to be written at a much higher level, without regard for the specific underlying transaction system. The low-level transaction system is totally abstracted by the EJB container that runs behind the scenes.

The bean components are responsible for deciding when a transaction should begin and commit or abort. If things run efficiently, a commit should be invoked; otherwise an abort should be invoked.

It is the responsibility of the XAResource provider (FioranoMQ) to integrate itself seamlessly with the transaction in progress and allow for behind the scenes enlistment, delistment, and commits the transaction with the transaction propagation being managed by the application server.

This provides a much higher level of abstraction as compared to using an external transaction manager explained in the preceding section. In addition, it achieves the objective of isolating the "business logic designer"/"bean designer" from the details of how the transaction progresses. The bean no longer has to take care of specific enlisting, delisting, and commit or rolling back the transaction in progress as opposed to when the transaction is controlled using an external transaction Manager.

The javax.transaction.UserTransaction interface defines methods that allow applications to define transaction boundaries and explicitly manage transactions. The UserTransaction implementation additionally provides the application components -- servlets, JSPs, EJBs (with bean-managed transactions) -- with the ability to programmatically control transaction boundaries. EJB components can access UserTransaction through EJBContext using the getUserTransaction () method. Some of the methods specified in the UserTransaction interface include begin (), commit (), getStatus (), rollback () and setRollbackOnly (). The J2EE server provides the object that implements the javax.transaction.UserTransaction interface and makes it available through JNDI lookup.

FioranoMQ as a XAResource provider can be integrated with any J2EE application server so that it is successfully enlisted and performs in an expected manner when used with a UserTransaction object provided by the application server.

FioranoMQ XA Implementation Notes

Distributed transactions are supported for both topics as well as queues as long as the data store used can participate in distributed transactions. FioranoMQ supports the use of RDBMS as the data store for messages and this need to be used in XA scenarios. Fiorano's File based data store cannot be used for destinations that need to be used in XA.

FioranoMQ XA implementation does provide support for local transactions well. This implies that when a new XA session is created, it has a local transaction context. This local transaction context ends when the XA session gets associated with a global transaction context. XAResource associated with the session starts a new distributed transaction (XAResource.start). When the global transaction ends (XAResource.end() or XAResource.rollback()), xasession automatically switches back to local transaction context and the JMS session of the xa session can be used as a transacted JMS session, invoking its commit/rollback methods.

The behavior of createSession call on the XAConnection object is undefined in the JTA specifications.

Session session = xaConnection.createSession(boolean transacted, int ackMode);

boolean transacted: usage undefined.

int ackmode: usage undefined.

FioranoMQ behaves the following way when it encounters a createSession (or createTopicSession/createQueueSession) on the xaConnection (or XATopicConnection/XAQueueConnection) object.

It returns the session (or topicsession/queueSession) object whose behaviour is defined completely by the specified parameters. For example in the following call a transacted (non-xa) session object is returned, which can take part in the JMS transactions.

xaConnection.createSession(true, 0);

All the started transactions are rolled back on server startup/on application close.

If a FioranoMQ client terminates after a transaction is successfully ended, (XAResource.TMSuccess)/suspended (XAResource.TMSUSPEND)/ prepared (returned with an XAResource.XA_OK flag), then this transaction can be used from the same status (prepared, ended or suspended), when the client activates again. Similarly, if a FioranoMQ server terminates when any ended/suspended/prepared transactions were active, then after server startup, FioranoMQ clients remain unaffected and can continue working on the transaction.

RDBMS based topics, on which messages are published in a transaction, get locked in the prepare call. Hence, no other transaction can be prepared in which messages are being published on a locked topic. All Non-xa publishers publishing on a locked topic have to wait for the release of lock to publish a message.

LimitatiHons of XA Implementation of FioranoMQ

Transaction timeout APIs and forget() APIs of XAResource are not supported in this release.

A durable subscriber can be associated with only one distributed transaction at any point of time. It can be associated with the next transaction only when it is disassociated with the previous transaction. The durable subscriber gets associated with the transaction when the xaResource object, created from the same session object as the durableSubscriber, starts a new transaction. This association ends when the resource object commits/rollback the started transaction.

The following code snippet explains when the association of the durable subscriber ends with the transaction.

```
// Get the topic session object

TopicSession ts = xats.getTopicSession();

// create a durable subscriber. This subscriber is not associated with

// the distributed transaction as the distributed transaction has not started yet.

TopicSubscriber   subscriber =  ts.createDurableSubscriber(topic,"clientId");

// get the xaResource object

XAResource xaresource = xats.getXAResource();

// start the resource Object. During this call, above durable subscriber gets

// associated with the transaction.

xaresource.start(xid,XAResource.TMNOFLAGS);

Application perform some work here

// end the resource object. The association does not end when the resource ends

xaresource.end(xid,XAResource.TMSUCCESS);

// prepare the resource object

xaresource.prepare(xid);

// commit the resource object. The association ends when the resource object gets

// committed. Similarly the association ends when the resource objects rollbacks.

xaresource.commit(xid, false);
```

Following are the methods with which a durable subscriber can be disassociated from the distributed transaction.

Single phase commit: Committing the transaction in single phase disassociates the durable subscriber from the distributed transaction.

```
xaresource.commit(xid, true);
```

Two phase Commit: Committing the transaction in two phases disassociates the durable subscriber from the distributed transaction.

```
xaresource.prepare(xid);

xaresource.commit(xid, false);
```

Rolling back the transaction: Rolling back the transaction disassociates the durable subscriber from the distributed transaction.

```
xaresource.rollback(xid);
```

Invalid attempts: The association of the durable subscriber does not end when the resource object ends the distributed transaction. As explained above, the association ends only when the transaction commits/rollbacks the started transaction. In case a durable subscriber is associated with a distributed transaction, which has ended, performing any of the following functions leads to throwing of an exception by the FioranoMQ server.

Start transaction: FioranoMQ server throws an exception in case an attempt is made to start a new transaction when the previously started transaction has not committed/rolled back.

The following code snippet explains it more clearly.

```
// create a durable subscriber.

// TopicSubscriber   subscriber =  ts.createDurableSubscriber(topic,"clientId");

// start a resource Object.

xaresource.start(xid,XAResource.TMNOFLAGS);

 ..

// end the resource object.

xaresource.end(xid,XAResource.TMSUCCESS);

// Attempt to start a different transaction leads to an exception

xaresource.start(xid2,XAResource.TMNOFLAGS);
```

The application provider is advised to commit or rollback the ended transaction before starting a new one. The following code snippet explains the workaround to the mentioned problem.

```
// get the resource object

xaresource = xats.getResource();
```

The transaction should be committed before creating the durable subscriber. The transaction can be committed either in a single phase or two phases.

```
xaresource.prepare(xid);

xaresource.commit(xid,false);
```

or

```
xaresource.commit(xid,true);
```

In case the transaction cannot be committed, the transaction should be rolled back before creating the durable subscriber.

```
xaresource.rollback(xid);
```

Restart the application: FioranoMQ server retains the state of all the transactions that have ended or that have been prepared. In case an application crashes after ending the transaction, it can restart/prepare/commit/rollback the transaction. In case the application in which a durable subscriber is associated with the ended transaction crashes, attempting to use the subscriber in non-xa transaction or some different transaction leads to an exception.

Consider an example in which a durable subscriber, dsubscriber, is associated with a distributed transaction, xid. The application after ending the transaction xid, crashes. When the application restarts, the following leads to an exception:

Subscriber creation without starting the transaction: An attempt to create a subscriber, without starting the ending transaction leads to an exception thrown by the JMS server.

```
// get the resource object

xaresource = xats.getResource();

// attempting to create a durable subscriber, without starting the transaction leads to

//an exception.

dsubscriber = ts.createDurableSubscriber(topic, "subId");
```

The application provider is advised to commit/rollback/restart the ended distributed transaction before creating the durable subscriber.

```
// get the resource object

xaresource = xats.getResource();
```

The transaction should be committed before creating the durable subscriber. The transaction can be committed either in single phase or two phases.

```
xaresource.prepare(xid);

xaresource.commit(xid,false);
```

Or

```
xaresource.commit(xid,true);
```

In case the transaction cannot be committed, the transaction should be rolled back before creating the durable subscriber.

```
 xaresource.rollback(xid);
```

In case the transaction needs to be restarted, it should be restarted before creating the durable subscriber

```
 xaresource.start(xid, XAResource.TMJOIN);

// Attempt to create the durable subscriber works perfectly fine after this

dsubscriber = ts.createDurableSubscriber(topic, "subId");

// attempting to create a durable subscriber, without starting the transaction leads to an exception.

dsubscriber = ts.createDurableSubscriber(topic, "subId");
```

Distributed Transactions does not work for unified connections.

Note: If a transaction is in the started phase, then it would rollback automatically, if the application or server terminates.

Chapter 18: FioranoMQ Content Based Routing

Introduction

Routing and addressing a message on communication networks is traditionally handled by specialized addressing and routing information attached to, or otherwise associated with the message. Today's applications typically require a message to be addressed and routed according to the contents of the message. The pervasiveness of XML helps define the new content-based addressing and routing problem as fast consumer selection for XML messages, using X Path predicates and SQL 92 syntax. High-speed evaluation and selection algorithms, coupled with high-speed message delivery systems, are required to fix this issue. FioranoMQ Content-based Routing (CBR) combines the world's fastest JMS server with ultra high-speed proprietary routing algorithms to provide scalable and fast content-based routing.

FioranoMQ Content Based Routing

Fiorano Software is the first vendor to introduce an intelligent, standards-based message, content-based routing system. It provides organizations greater flexibility in achieving the ultimate aim of empowering consumers with the timely information they require.

This chapter starts with an introduction, outlines the content-based routing (CBR) problem that FioranoMQ CBR solves, a description of how to use CBR with samples, includes a description of the types of XML currently supported, a section covering the entire subset of XPath supported and a review of SQL 92 syntax for predicates. It is assumed that the reader has a working knowledge of JMS. If not, FioranoMQ documentation can be found in the / fmq / docs folder of the FioranoMQ installation directory and the JMS 1.0.1 API specification can be found at:

`http://java.sun.com/products/jms/javadoc-102a/index.html`

Current pub/sub systems are group-based. In these systems, messages are classified as belonging to a certain group, referred to as Topic (also known as channels). Publishers are required to label each message with a topic name, while consumers subscribe to all messages on a particular topic. For example, a topic based pub/sub system for stock trading may define a message header field for each issue. Publishers post messages after labeling them, by setting a header property to the particular issue. Subscribers can set preferences based on predetermined header properties, known as message selectors.

This type of approach has the following drawbacks:

- Requires high processing overhead on the publisher when setting message header properties appropriately.
- Limits the scope of messages to a domain specific set because the selectors for the subscribers have to be in accordance with the message header properties that have been set. Thus there is loss of flexibility in the messages to change domains.

The next generations of pub/sub systems offer a better alternative to group-based systems, known as content-based routing. These systems route messages to subscribers, based on content instead of message properties contained in the headers. There is no overhead imposed on the publisher and prior knowledge of the domain is not required. Subscribers have the added flexibility of choosing filtering criterion along multiple dimensions, without requiring pre-definition of groups.

In a stock trading example for group based systems, the subscribers can only select trades by issue name. In contrast, the content-based subscriber is free to use an orthogonal criterion, such as volume; or a collection of criteria, such as issue, price and volume. In addition, management of content-based systems is simpler, as there is no administration overhead required to pre-define and maintain the groups. Content-based routing systems allow for more efficient and scalable message distribution systems.

Let us continue with the stock trade example. Consider a brokerage firm that may have thousands of subscribers interested in information on stock trades. Each subscriber has its own selection criterion based on its requirement. One subscriber would like to be alerted when two stocks fall below a certain price, for example:

MSFT stock falls below 55 AND ORCL stock falls below 15

Another subscriber would like to be alerted when any one of three stocks, changes its price and when the market volume exceeds a certain threshold, for example:

```
(INTC < 21 OR CSCO > 20 OR GE > 32) AND DowVolume > 1,000,000,000
```

In an unrelated instance, a subscriber would like to be alerted when a Formula 1 driver sets a new lap record at Imola, for example:

```
(Car_Make = Ferrari) AND (Driver = Schumacher) AND (Circuit = Imola) AND (Lap Time < 1.01 mins)
```

The information a subscriber may be interested in is unlimited. Messages, events and alerts may be desired for items such as changes in inventory, new Purchase Orders, product delivery, receipt of a Request For Quote and late breaking news.

In order to efficiently implement a pub/sub system with content-based routing, one must first find an efficient solution to the problem of matching a message against a large number of subscribers, referred to as the matching problem. In addition, there must be an efficient and easy to understand standards based language, by which the subscribers register and store their personal message preferences. FioranoMQ CBR effortlessly provides these functionalities.

Using FioranoMQ Content Based Routing

This section explains how to use FioranoMQ Content Based Routing (CBR) with samples and a description of the currently supported XML syntax. FioranoMQ CBR is based on the JMS 1.1 specifications with extensions to support content-based routing.

Setting up the FioranoMQ Server for CBR

The Content Based Routing support of FioranoMQ is available only in the Publish/ Subscribe JMS domain and not in the PTP (Point to Point) domain. By default, the CBR support is not enabled on the server. It can be enabled using the Fiorano Studio. Perform the following steps to set up the FioranoMQ server for CBR:

1. Start the Fiorano Studio.
2. Select **Tools** > **Configure Profile** from the menu bar, select the profile directory in the resulting Select Profile Directory dialog box and click the Open button. This shifts the FioranoMQ environment to the offline mode.
3. Now, navigate to **FioranoMQ** > **Fiorano**> **etc**> **FMQConfigLoader** in the Profile Explorer pane. The properties are displayed in the Properties pane.

4. Select the property named **UseFioranoCbr** and set its value to True from the drop-down list.
5. Right-click the root node in the profile explorer and select the Save option from the resulting shortcut menu.

Now the server has CBR enabled, and clients can send and receive XML messages with XPath selectors. More information on the client side changes is available in the subsequent sections.

FioranoMQ CBR XPath Support

FioranoMQ CBR utilizes a subset of XPath notation and SQL92 syntax to specify XPath message selectors.

Essentially, you must use only absolute paths. You can combine several XPath string with AND/OR conditions. For example, provide an XML as follows:

```
<Sports>
        <Soccer>
                <Team>Manchester United</Team>
         <Record>33,5,1</Record>
         <LastGame>
                        <Team>Arsenal</Team>
                        <Date>4/5/01</Date>
                        <HomeAway>Home</HomeAway>
                        <HomeScore>2</HomeScore>
                        <VisitorScore>0<VisitorScore>
         </LastGame>
 </Soccer>
</Sports>
```

In the above example, if you only need to view messages for Manchester United Soccer, then use the following XPath message selector:

```
"/Sports/Soccer/Team = 'Manchester United'"
```

In the above example, if you only need to view messages where Manchester United was the home team, then register the following XPath message selector:

```
"/Sports/Soccer/Team = 'Manchester United'
and /Sports/Soccer/LastGame/HomeAway = 'Home'"
```

Publishing XML Messages

For the Publisher sample to use CBR, set the value of InitialContext environment variable UseFioranoCbr to true. To use CBR related proprietary APIs, the package fiorano.jms.runtime.xpubsub have to be imported. XML messages are sent as contents of FioranoXMLMessages.

FioranoXMLMessages in JMS terminology is a TextMessage and can be created using FioranoTopicSession.

Following is a code snippet for creating a publisher to publish XML messages.

```
 /**
 * @(#)FCRPublisher.java          1.0, 04/25/2001
 *
 * Publishes 3 xml messages on the topic specified by client.
 * The mode of delivery can be both persistent and non-persistent.
 * One of these messages would be received by the XCRSubscriber sample.
 *
 * Copyright (c) 2001 by Fiorano Software, Inc.,
 * Los Gatos, California, 95030, U.S.A.
 * All rights reserved.
 *
 * This software is the confidential and proprietary information
 * of Fiorano Software, Inc. ("Confidential Information").  You
 * shall not disclose such Confidential Information and shall use
 * it only in accordance with the terms of the license agreement
 * enclosed with this product or entered into with Fiorano.
 */
import javax.jms.*;
import java.util.*;
import javax.naming.*;
import fiorano.jms.services.msg.def.*;
import fiorano.jms.runtime.xpubsub.*;
import fiorano.jms.runtime.naming.FioranoJNDIContext;

public class FCRPublisher
{
    public static void main( String args[] ) throws Exception
    {
        Hashtable env = new Hashtable();
        env.put (Context.SECURITY_PRINCIPAL, "anonymous");
        env.put (Context.SECURITY_CREDENTIALS, "anonymous");
        env.put (Context.PROVIDER_URL,
                  "http://localhost:1856"  );
        env.put (Context.INITIAL_CONTEXT_FACTORY,
                    "fiorano.jms.runtime.naming.FioranoInitialContextFactory");

        InitialContext ic = new InitialContext (env);
        TopicConnectionFactory tcf =
                        (TopicConnectionFactory) ic.lookup ("primaryTCF");

        TopicConnection topicConnection = tcf.createTopicConnection();

        Topic topic = (Topic)ic.lookup("primaryTopic");
        FioranoTopicSession topicSession =
(FioranoTopicSession)topicConnection.createTopicSession(false,1);
        TopicPublisher topicPublisher = topicSession.createPublisher(topic);
```

```
        System.out.println("Starting message delivery ....");

        FioranoXMLMessage tMsg1 = topicSession.createXMLMessage();
        FioranoXMLMessage tMsg2 = topicSession.createXMLMessage();
        FioranoXMLMessage tMsg3 = topicSession.createXMLMessage();


        String[] sym = {"MSFT","IBM","HWP"};
        int[] price = {40,50,60};
        String[] xml = new String[3];

        for(int i=0;i<3;i++)
        {
            xml[i] = "<quote>";
            xml[i] = xml[i] +  "<symbol>"+ sym[i] +"</symbol>";
            xml[i] = xml[i] +  "<askprice>"+ price[i] +"</askprice>";
            xml[i] = xml[i] +  "</quote>";
        }
        try
        {
            // setting the xml to the text messages
            tMsg1.setText(xml[0]);
            tMsg2.setText(xml[1]);
            tMsg3.setText(xml[2]);

            tMsg1.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);
            tMsg2.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);
            tMsg3.setJMSDeliveryMode (DeliveryMode.NON_PERSISTENT);

            topicPublisher.publish(tMsg1);
            System.out.println("Published the message :: " + xml[0]);
            topicPublisher.publish(tMsg2);
            System.out.println("Published the message :: " + xml[1]);
            topicPublisher.publish(tMsg3);
            System.out.println("Published the message :: " + xml[2]);
        }
        catch( Exception e )
        {
            System.out.println("Exception in publishing:" +e );
        }
    }
}
```

Subscribing to XML Messages

The creation of a FioranoMQ CBR subscriber is similar to creating a JMS subscriber. In addition, set the value of InitialContext environment variable UseFioranoCbr to true. The subscribers (durable and non-durable) can then be created in a similar manner as in plain PubSub. The message selector must be of the form as explained in the section "Error Reference source not found." . The message can be any valid XML, passed as contents of FioranoXMLMessages.

CreateDurableSubscriber

The following API creates a Durable Subscriber, which only receives messages if they confirm to the specified Xpath Message Selector string:

```
 /**
 * Fiorano's proprietary API to create XPath Durable Subscriber
 *
 * This API creates an XPath Durable Subscriber.
 *
 * @param topic - the non temporary topic to subscribe to
 * @param subscriptionID - ID used to identify this subscription
 * @param messageSelector - string containing the XPth message selector or normal JMS Message
selector
 * @param NoLocal - if set, inhibits delivery of messages published by its own connection
 * @exception JMSException if operation fails
 *
 */
public TopicSubscriber createDurableSubscriber (

                                Topic topic,
                                String subscriberID,
                                String messageSelector,
                                boolean noLocal)
                       throws JMSException ;
```

CreateSubscriber

The following API creates a Non-Durable Subscriber, which only receives messages if they pass the specified Xpath Message Selector string:

```
    /**
     * Fiorano's proprietary API to create XPath Subscriber
     *
     * Create XPath Durable Subscriber
     *
     * @param topic - the non temporary topic to subscribe to
     * @param messageSelector - string containing the XPth message selector or normal JMS
Message selector.
     * @param NoLocal - if set, inhibits delivery of messages published by its own connection.
     * @exception JMSException if operation fails
     */
     public TopicSubscriber createSubscriber (

                                Topic topic,
                                String messageSelector,
                                boolean noLocal)
                 throws JMSException  ;
To create a FioranoMQ CBR subscriber, you can follow the example code snippet given below, which
sets up an XPath Subscriber, onMessage and onException listeners.
/*
 * Copyright (c) 2001, Fiorano Software, Inc.
 * All Rights Reserved
 *
 * FileName   : XPathSubscriber.java
 *
```

```
 *
 * Implements an asynchronous listener to listen
 * for messages published on the topic - "primaryTopic"
 * only receiving message which match the selector parameter
 *
 * Questions/comments/suggestions?
 * Please visit: http://www.fiorano.com
 * Or e-mail: support@fiorano.com
 *
 * @since     FioranoMQ 6.0, August 2002
 */

import javax.jms.*;
import javax.naming.*;
import java.io.*;
import java.util.*;
import fiorano.jms.services.msg.def.*;
import fiorano.jms.rtl.*;
import fiorano.jms.runtime.naming.FioranoJNDIContext;

import java.net.*;

class Subscriber implements MessageListener,ExceptionListener
{
    public static void main (String args[])
    {
       Subscriber subscriber = new Subscriber ();
        try
        {

            // 1. Create the InitialContext Object used for looking up
            //     JMS administered objects on the Fiorano/EMS
            //     located on the default host.
            //
            Hashtable env = new Hashtable ();
            env.put (Context.SECURITY_PRINCIPAL, "anonymous");
            env.put (Context.SECURITY_CREDENTIALS, "anonymous");
            env.put (Context.PROVIDER_URL,
                     "http://localhost:1856");
            env.put (Context.INITIAL_CONTEXT_FACTORY,
                     "fiorano.jms.runtime.naming.FioranoInitialContextFactory");
           InitialContext ic = new InitialContext (env);
            System.out.println ("Created InitialContext :: " + ic);

            // 1.1 Lookup Connection Factory and Topic names
            //
            TopicConnectionFactory tcf =
                        (TopicConnectionFactory) ic.lookup ("primaryTCF");
            Topic topic = (Topic)ic.lookup("primaryTopic");


            // 2. create and start a topic connection
            System.out.println("Creating topic connection");
            TopicConnection topicConnection = tcf.createTopicConnection();
        // Register an Exception Listner
        topicConnection.setExceptionListener (subscriber);
```

```
            topicConnection.start ();

            // 3. create topic session on the connection just created
            System.out.println("Creating topic session: not transacted, auto ack");
            TopicSession topicSession = topicConnection.createTopicSession(false,1);

            // 4. create XpathSubscriber
            System.out.println("Creating topic, subscriber");
            String selector = "/quote/symbol = 'IBM' and /quote/askprice > 40";
            TopicSubscriber topicSubscriber =
              topicSession.createSubscriber(topic,selector,false);

            // 5. install an asynchronous listener/callback on the subscriber object
            //    just created
            System.out.println ("Ready to subscribe for messages :");
            topicSubscriber.setMessageListener (new Subscriber ());
        }
        catch (Exception e)
        {
            e.printStackTrace ();
        }
    }

    /**
     * Message listener which receives messages aysynchronously
     * For the bound subscriber.
     */
    public void onMessage( Message m )
    {
        if( !(m instanceof FioranoXMLMessage) )
        {
            return;
        }

        String s = "";
        try
        {
            s = ((FioranoXMLMessage)m).getText();
        }
        catch( Exception e )
        {
            System.out.println("Exception in getText() :" + e );
        }
        System.out.println ("Received the message :: "+s);
    }
    /**
     * If a JMS provider detects a serious problem with a
     * Connection this method is invoked passing it a JMSException
     * describing the problem.
     * @param JMSException e
     */
    public void onException (JMSException e)
    {
//Report the Error and take necessary Error handling measures
        String error = e.getErrorCode ();
System.out.println (error);
```

```
        ((fiorano.jms.common.FioranoException)e).printCompleteStackTrace();
    }
}
```

More samples of using FioranoXCR content-based routing are available in the samples directory of the FioranoMQ installation under pubsub / ContentBasedRouting.

Handling Massive Number of Subscribers

In an environment where millions of subscribers are required, you can cascade FioranoMQ servers in such a form that your data source publishes to multiple FioranoMQ servers simultaneously, each source publishing or sending messages to, say, 100 FioranoMQ servers. Each of these in turn sends information to another set of FioranoMQ servers, or directly send to the system that ultimately transfers information to the subscribers. For best results, it is recommended not to run more than 500 subscribers on each FioranoMQ connection.

Following are the key concepts to handle large number of subscribers:

- Keep XML messages small (do not mix domain data, such as weather and sports, and do not add information on more than one item within a domain at a time, such as stock quotes for more than 1 company).
- Use fewer connections and more subscribers per connection.
- Use suitably powered systems that run the subscribers so that they do not decelerate the movement of messages. Most slowdowns occur due to the subscribers that are unable to quickly process the messages.
- Segregate domain data on different topics (for example, use a topic for weather and a different topic for sports).

The illustration in Figure 4-1 depicts one such environment, showing incoming XML messages distributed to several FioranoMQ boxes. These FioranoMQ boxes in turn feed the distribution systems, which ultimately send message to devices, such as cell phone and wireless PDAs. This example illustrates how multiple subscribers can be used and data is distributed to different topics.

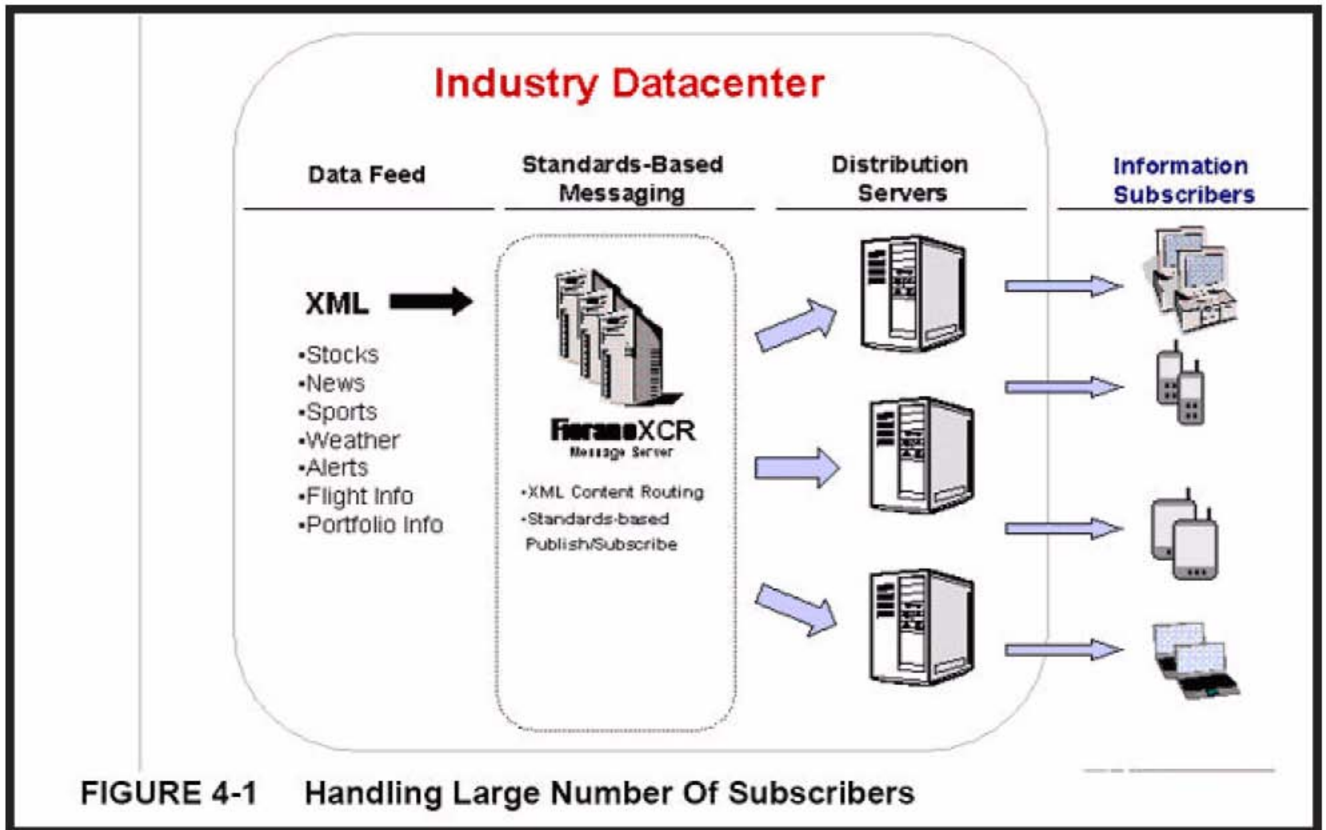


FIGURE 4-1 Handling Large Number Of Subscribers

XML Support

The FioranoMQ Content Based Routing supports elements and attributes type XML. The subsequent version of FioranoMQ supports any XML file. However, it is highly recommended that the XML files be atomic in nature such that they can be parsed and delivered quickly. Large XML files takes longer to parse and distribute. Additionally, when formatting the XML, several methods are available to package the information. This implies that information can be packaged in an Element only, Attributes only or Elements and Attributes xml. Depending on the packaging, it can be optimized for CBR or non-optimized for the same.

For example, if a subscriber wants to receive stock information of company XYZ, and an XML file contains stock information of both company XYZ and company ABC. Then the subscriber receives, in essence, some information it does not require or need. Hence, when disseminating stock quotes, it is preferable to add information pertaining to one stock in a single message.

This accomplishes the following two issues:

- It keeps the message small
- Unwanted content are not sent to the subscribers

For the purpose of discussion here, we can break down XML into three categories, elements only, attributes only and elements and attributes. Examples of each type of XML, as well as example message selectors can be found in the following section. When wrapping data in XML, to be optimized for content-based routing, it is recommended that the elements only and/or attributes only types be used for highest message throughput.

An example of elements only is as follows:

```
<atag>
 <btag>value</btag>
 <ctag>value</ctag>
     .
     .
     .
<ztag>value</ztag>
</atag>
```

In the above example, the level of nesting is not fixed to two, but shows that this XML type contains only elements. Processing this type of XML is the fastest.

An example of attributes only is as follows:

```
<Stock quotes>
 <Quote Symbol="x" ASkPrice="y" BidPrice="z">
 <Quote Symbol="a" AskPrice="b" BidPrice="c">
</Stock quotes>
```

The above example shows that there are only attributes and contains no elements, which can be referenced. This does not imply that an elements only XPATH, such as "/Stock quotes/Quote Symbol IS NOT NULL," cannot be used. This selector indicates that any XML message containing this element is selected. Processing this type of XML is a little slower than the elements only type.

An example of elements and attributes is as follows:

```
<atag>
 <btag att1="x" att2="y">
   <ctag att3="z">value</ctag>
 </btag>
</atag>
```

The above example shows how elements can be found within the elements containing attributes. Processing this type of XML can be much slower than either of the other two types, depending on size of the message. It would also require more complex XPath identifiers.

FioranoMQ Content - Based Message Selector Language

This section explains SQL92 syntax predicates and the subset of XPath supported by FioranoMQ CBR. In addition, it explains the type of XMLs that are supported along with suitable example XMLs and message selectors.

General Form

Message selection criterion is registered by consumers using the FioranoMQ Content-based Message Selector (CbMS) language, which is a subset of the SQL92 conditional expression syntax. It is combined with certain XPath like notation for retrieving identifiers and values from XML documents.

The order of evaluation of a CbMS is from left to right within precedence level. Parentheses can be used to change this order. Predefined selector literals and operator names are written in uppercase; however, they are case insensitive.

XPath notation is used as a reference mechanism to return one or many elements from an XML file. The entire XPath specification is sometimes inaccurate for content-based message selection, particularly where speed and scalability is essential. XPath notation must return only a single value, else an exception is launched.

The general form of CbMSs is as follows:

```
[/ {not} C {[in]|[between]} q y {and|or} /]
```

Where:

"[/ /] One or many

"{} Optional

"{|} Optional, and if used then select one C Identifier: For details refer to the section Identifiers.

"q Operator. For details refers to the section Operators.

"y Literal. For details refer to the section Literals .

Examples:

Following are a few examples of FioranoMQ content based message selector language:

```
/quotes/symbol = 'IBM' (NOTE: C is '/quotes/symbol', q is '=' and y is 'IBM')
/quotes/symbol='IBM' or /quotes/symbol='CSCO'
(/quotes/symbol='IBM' or /quotes/symbol='CSCO') and /quotes/bidprice>150
/quotes/symbol in ['IBM','CSCO'] and (/quotes/pricedelta between 1 and 10)
```

Subset of Supported XPath Queries

The current limitations of CbMS XPath support as compared to the full XPath specification are as follows:

1. **Only absolute paths are allowed**.

   This indicates that all the XPath strings must start with '/'. '//' is not supported for relative paths. All the paths must be specified from the root of the XML. For example, the following XPath query is invalid:

"//person"

   If a person is found under parent /researchers, then the correct XPath query would be as follows:

"/researchers/person"

2. **A test node can have only one predicate**.

   This indicates that a query as follows is invalid, even though it is valid in XPath:

"//researchers/person[@name = "Shin"][@loc = "Bethesda"]"

   However, the following query is acceptable:

"/researchers/person[@name = "Shin"] and /researchers/person[@loc = "Bethesda"]"

3. **A predicate cannot appear inside another predicate.**

   This indicates that a query as follows is invalid, since it has a nested predicate inside another predicate.

"/researchers/person[@name = /salesperson/person/name[@id > "8080"]])"

4. **The character '|' and the expressions 'and' and 'or' are not supported inside a predicate**.

   This implies that the operator "|" (Union operator) is not allowed for connecting path expressions outside predicates. This is accomplished through the use of the 'OR' keyword. Inside predicates, "|", "and" and "or" operators are not allowed. Hence, the following query is invalid:

"/a[@id="1" or in("2","3")]"

   Thus, represent this query as follows:

"/a[@id="1"] or /a[@id=2] or /a[@id=3]"

5. **One side of the equality and relation operator (=, <, >, <=, >=) must be a literal**.
6. **Join operation is not supported.**

   Points (5) and (6) indicate that comparison inside predicates is limited. At present, one argument must be literal for the equality and comparison operator. For instance, the following query is valid:

"//a[@id > "100"]" or "//a[title = "XPERT"]"

   Whereas, the following query is invalid:

"//a[@id=//b/@id]"

   At present, semi-join is not allowed.

7. **Only three functions, "contains()", "in()", and "between()" are supported**.

   This indicates that only three functions, "contains()", "in()", and "between()" are supported.

8. **"*" representing 'anything' is not supported**.

   This indicates that the wildcard character "*" is not supported in any form.

Identifiers

Identifiers are expressed in terms of XPath notation. XPath can be used to refer to any part or parts of an XML document. Since message selection is based on actual contents of a message and not only the presence of the message, the XPath notation must return a single attribute of an element in the XML file. Any valid XPath notation is supported, but in case an XPath returns multiple elements, only the first one is evaluated. It is highly recommended that only the following XPath notation be used:

"/ROOT/CHILD where CHILD contains only 1 element

"/ROOT/CHILD[x] where x is the number of exact element of type CHILD

"/ROOT/CHILD[last()] selects the last element of type CHILD

"/ROOT/CHILD[first()] selects the first element of type CHILD

Operators

XPATH selectors involve the use of various arithmetical, logical and conditional operators. FioranoMQ CBR supports most of the operators. This section explains, with examples, the support of following operators:

Standard bracketing () is supported. This implies that the conditions inside a bracket are evaluated first and then the outer conditions are evaluated.

"Logical operators in precedence order: NOT, AND, OR

"Comparison operators: =, >, >=, <, <=, <>

Note - Not equal (<>) is supported internally as (NOT (C = y))

"Only like type values can be compared, for example, a string can be compared to a string and a boolean to a boolean. However, it is valid to compare exact numeric values and approximate numeric values (the type conversion required is defined by the rules of Java numeric promotion). If the comparison of non-like type values is attempted, then the selector is always false.

"String and Boolean comparison is restricted to = and <>. Two strings are equal if and only if they contain the same sequence of characters.

Note - Not equal (<>) is supported internally as (NOT (C = y))

"-+, - unary

"*, / multiplication and division

"+, - addition and subtraction

"Arithmetic operations must use Java numeric promotion

"{NOT} BETWEEN arithmetic-expr1 and arithmetic-expr2 (exact numeric values only)

Note - BETWEEN x and y is supported internally as z >= x AND z <= y

NOT BETWEEN x and y is supported internally as z < x OR z > y

" {NOT} IN (string-literal1, string-literal2...)

Note - IN (x,y,z) is supported internally as w = x OR w = y OR w = z

Note - NOT IN (x,y,z) is supported internally as ((NOT(w = x)) AND (NOT(w = y)) AND (NOT(w = z))))

"(OPTIONAL) {NOT} LIKE pattern-value [ESCAPE escape-character] comparison operator, where identifier has a String value. Pattern-value is a string literal, where '_' represents any single character; '%' represents any sequence of characters (including the empty sequence); and all other characters represent themselves. The optional escape-character is a single character string literal, whose character is used to escape the special meaning of '_' and '%' in pattern-value.

Examples:

Following are few examples of the use of LIKE operator with the results:

"Phone LIKE '12%3' is true for '123' '12993' and false for '1234'

"Word LIKE 'l_se' is true for 'lose' and false for 'loose'

"Underscored LIKE '_%' ESCAPE '\' is true for '_foo' and false for 'bar'

"Phone NOT LIKE '12%3' is false for '123' and '12993' and true for '1234'

"If identifier of a LIKE or NOT LIKE operation is NULL, the value of the operation is unknown.

"{NOT} NULL allows for testing of the existence of an element or elements within an XML without actually testing any values.

Examples:

/quotes/bidprice NOT NULL succeeds for any XML which has any elements matching /quotes/bidprice
/quotes/bidprice. NULL succeeds for any XML which has no elements matching /quotes/bidprice

Literals

A string literal is enclosed in single quotes with an included single quote, represented by doubled single quote such as 'literal' and 'literal''s'. Similar to Java String literals, these use the unicode character encoding.

An exact numeric literal is a numeric value without a decimal point such as 57, -957, +62; numbers in the range of Java long are supported. Exact numeric literals use the Java integer literal syntax.

An approximate numeric literal is a numeric value in scientific notation such as 7E3, -57.9E2 or a numeric value with a decimal such as 7., -95.7, +6.2; numbers in the range of Java double are supported. Approximate literals use the Java floating point literal syntax.

There are only 2 possible boolean literals, TRUE and FALSE.

Example XMLs

Complete XPath specification can be found at :

http://www.w3.org/TR/xpath

The following are three examples of XML files and associated XPath strings that are found within the XML files. For each XML file there is an example message selector. After each sample message selector are the strings that need to be returned by the single pass parser. Following each string, is the value that is returned by the single pass parser, if value of the particular XPath string is found to be advantageous. If the value of the XPATH string is found matching in the parsed XML, then the message is selected.

Elements Only XML

```
<Stock quotes>
 <Symbol>csco</Symbol>
 <AskPrice>33.50></AskPrice>
 <BidPrice>33.25>>/BidPrice>
</Stock quotes>
```

Example 1 for Elements Only XML

Related XPath predicate is as follows:

```
/Stock quotes/AskPrice > 33.2 and /Stock quotes/Symbol = 'csco'
```

Description: Generate the XML if AskPrice is greater than 33.2 and Symbol is 'csco'. This is an example of using equality and an inequality check."

Example 2 for Elements Only XML

Related XPath predicate is as follows:

```
/Stock quotes/AskPrice > 33.2 and /Stock quotes/Symbol LIKE 'csc%'
```

Description: Generate the XML if AskPrice is greater than 33.2 and Symbol starts with 'csc'.

Following is an example of using Ineq and LIKE trees:

XPath parser would return:

```
/Stock quotes/Symbol
/Stock quotes/AskPrice
/Stock quotes/BidPrice
```

Values returned for each XPath string:

```
/Stock quotes/Symbol 'csco'
/Stock quotes/AskPrice '33.50'
/Stock quotes/BidPrice '33.25'
```

Attributes Only

```
<Stock quotes>
 <Quote Symbol>="csco" AskPrice="33.50" BidPrice=33.25">
 <Quote Symbol>="ibm" AskPrice="122.50" BidPrice="122.25">
<Stock quotes>
```

Related XPath predicate is as follows:

```
/Stock quotes/Quote[@Symbol='csco'] NOT NULL and /Stock quotes/Quote[@AskPrice] > 33.2
```

Description: Generate the XML if it contains a Quote Element with attribute Symbol set to 'csco' and an AskPrice attribute > 33.2. Following is an example of using Ineq and NULL trees. This example also shows the use of checking attribute values inside the XPath query.

XPath parser would return:

```
/Stock quotes/Quote[@Symbol]
/Stock quotes/Quote[@AskPrice]
/Stock quotes/Quote[@BidPrice]
/Stock quotes/Quote[@Symbol]
/Stock quotes/Quote[@AskPrice]
/Stock quotes/Quote[@BidPrice]
```

Values returned for each XPath string:

```
/Stock quotes/Quote[@Symbol]
'csco
/Stock quotes/Quote[@AskPrice]
'33.50'
/Stock quotes/Quote[@BidPrice]
'33.25'
/Stock quotes/Quote[@Symbol]
'ibm'
/Stock quotes/Quote[@AskPrice]
'122.50'
/Stock quotes/Quote[@BidPrice]
```

```
'122.25'
```

Elements and Attributes XML

```
<Quotes>
 <QuoteType Type='Quote of the Day'>

  'This is the funny quote of the day for April 2, 2001

 </QuoteType>
 <QuoteType Type='Stock Quote'>
  <Symbol>csco</Symbol>
  <AskPrice>33.50</AskPrice>
  <BidPrice>33.25</BidPrice>
</QuoteType>
```

Related XPath predicate is as follows:

```
/Quotes/QuoteType[@Type='StockQuote']/Symbol = 'csco' and
/Quotes/QuoteType[@Type='StockQuote']/AskPrice > 33.2
```

Description: Generate the XML if it contains QuoteType element with attribute Type = 'StockQuote' and elements = 'csco' and > 33.2. The following example uses Eq and Ineq trees:

XPath parser would return:

```
/Quotes/QuoteType[@Type='Quote of the Day']
/Quotes/QuoteType[@Type='Stock Quote']/Symbol
/Quotes/QuoteType[@Type='Stock Quote']/AskPrice
/Quotes/QuoteType[@Type='Stock Quote']/BidPrice
```

Values returned for each XPath String are as follows:

```
/Quotes/QuoteType[@Type='Quote of the Day']
         'This is the funny quote of the day for April 2, 2001'
/Quotes/QuoteType[@Type='Stock Quote']/Symbol
         'csco'
/Quotes/QuoteType[@Type='Stock Quote']/AskPrice
         '33.50'
/Quotes/QuoteType[@Type='Stock Quote']/BidPrice
         '33.25'
```

Limitations of FioranoMQ Content Based Routing

Following are a few known limitations of FioranoMQ Content Based Routing:

1. In case an attribute or a tag does not exist and a message selector attempts to check the presence of that attribute or tag, then the message is not selected, irrespective of the nature of the selector. For instance, consider the following XML:

```
<Stock quotes>
<Symbol>csco</Symbol>
<AskPrice>33.50</AskPrice>
<BidPrice></BidPrice>
</Stock quotes>
```

   A message selector of the following form would evaluate to false:

```
//Stock quotes/Symbol/AskPrice[@att1='0']=33.50
//Stock quotes/Symbol/AskPrice[@att1<>'0']=33.50
```

   This is because no attribute by the name 'att1' exists in the XML message and the message is not delivered.

   Similarly, a selector of the following form would also evaluate to false and that message shall not be delivered:

```
//Stock quotes/Symbol/BidPrice<>212
```

2. 'Between' is not supported in the case of attributes; however, it is supported in case of tags.

Consider the following XML:

```
<Stock quotes>
<Symbol>csco</Symbol>
<AskPrice att1='12'>33.50</AskPrice>
<BidPrice att2='13'>33.25</BidPrice>
</Stock quotes>
```

A selector of the following form is not supported;

```
/Stock quotes/Symbol/AskPrice[@att1 between '12' and '13']:
```

However, the following form is supported:

```
/Stock quotes/Symbol/AskPrice between 30 and 40
```

3. Message selectors involving mathematical operations are not supported. For example, consider the following XML:

```
<Stock quotes>
<Symbol>csco</Symbol>
<AskPrice>33.50</AskPrice>
<BidPrice>33.25</BidPrice>
</Stock quotes>
```

   A message selector of the following form is not supported:

```
/Stock quotes/Symbol/AskPrice = 0.3350*100
```

4. JMS message selectors of the following type are not allowed:

   Consider a TextMessage on which the following properties have been set:

```
textMessage.setIntProperty("low_val",10);
textMessage.setIntProperty("high_val",100);
textMessage.setIntProperty("mid_val",50);
```

A message selector of the following form is not supported:

```
mid_val between low_val and high_val
```